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:

An Analysis of Human-in-the-loop Approaches for Binary Analysis Automation

Thèse présentée et soutenue à Biot, le 25/03/2022, par

ALESSANDRO MANTOVANI

Abstract

In system and software security, one of the first criteria before applying an analysis methodology is to distinguish according to the availability or not of the source code. When the software we want to investigate is present in binary form, the only possibility that we have is to extract some information from it by observing its machine code, performing what is commonly referred to as *Binary Analysis* (BA). The artisans in this sector are in charge of mixing their personal experience with an arsenal of tools and methodologies to comprehend some intrinsic and hidden aspects of the target binary, for instance, to discover new vulnerabilities or to detect malicious behaviors.

Although this human-in-the-loop configuration has been well consolidated over the years, the current explosion of threats and attack vectors such as malware, weaponized exploits, etc. implicitly stresses this binary analysis model, demanding at the same time for high accuracy of the analysis as well as proper scalability over the binaries to counteract the adversarial actors. Therefore, despite the many advances in the BA field over the past years, we are still obliged to seek novel solutions.

In this thesis, we take a step more on this problem, and we try to show what current paradigms lack to increase the automation level. To accomplish this, we isolated three classical binary analysis use cases, and we demonstrated how the pipeline analysis benefits from the human intervention. In other words, we considered three human-in-the-loop systems, and we described the human role inside the pipeline with a focus on the types of feedback that the analyst "exchanges" with her toolchain. These three examples provided a full view of the gap between current binary analysis solutions and ideally more automated ones, suggesting that the main feature at the base of the human feedback corresponds to the human ability at comprehending portions of binary code.

This attempt to systematize the human role in modern binary analysis approaches tries to raise the bar towards more automated systems by leveraging the human component that, so far, is still unavoidable in the majority of the scenarios. Although our analysis shows that machines cannot replace humans at the current stage, we cannot exclude that future approaches will be able to fill this gap as well as evolve tools and methodologies to the next level. Therefore, we hope with this work to inspire future research in the field to reach always more sophisticated and automated binary analysis techniques.

Résumé

En matière de sécurité des systèmes et des logiciels, l'un des premiers critères avant d'appliquer une méthodologie d'analyse est de distinguer selon la disponibilité ou non du code source. Lorsque le logiciel que nous voulons investiguer est présent sous forme binaire, la seule possibilité que nous avons est d'en extraire des informations en observant son code machine, en effectuant ce qui est communément appelé *Binary Analysis* (BA). Les acteurs de ce secteur sont chargés de mêler leur expérience personnelle à un arsenal d'outils et de méthodologies pour comprendre certains aspects intrinsèques et cachés du binaire cible, par exemple pour découvrir de nouvelles vulnérabilités ou détecter des comportements malveillants.

Bien que cette configuration humaine dans la boucle se soit bien consolidée au fil des ans, l'explosion actuelle des menaces et des vecteurs d'attaque tels que les logiciels malveillants, les exploits armés, etc. met implicitement à l'épreuve ce modèle de BA, exigeant en même temps une grande précision de l'analyse ainsi qu'une évolutivité appropriée des binaires pour contrer les acteurs adverses. C'est pourquoi, malgré les nombreux progrès réalisés dans le domaine de la BA au cours des dernières années, nous sommes toujours obligés de chercher de nouvelles solutions.

Dans cette thèse, nous faisons un pas de plus sur ce problème et nous essayons de montrer ce qui manque aux paradigmes actuels pour augmenter le niveau d'automatisation. Pour ce faire, nous avons isolé trois cas d'utilisation classiques de l'analyse binaire et nous avons démontré comment l'analyse en pipeline bénéficie de l'intervention humaine. En d'autres termes, nous avons considéré trois systèmes "human-in-the-loop" et nous avons décrit le rôle de l'homme dans le pipeline en nous concentrant sur les types de feedback que l'analyste "échange" avec sa chaîne d'outils. Ces trois exemples nous ont fourni une vue complète de l'écart entre les solutions actuelles d'analyse binaire et les solutions idéalement plus automatisées, suggérant que la principale caractéristique à la base du retour d'information humain correspond à la compétence humaine à comprendre des portions de code

binaire.

Cette tentative de systématisation du rôle de l'homme dans les approches modernes de l'analyse binaire tente d'élever la barre vers des systèmes plus automatisés en tirant parti de la composante humaine qui, jusqu'à présent, est toujours inévitable dans la majorité des scénarios d'analyse binaire. Bien que notre analyse montre que les machines ne peuvent pas remplacer les humains au stade actuel, nous ne pouvons pas exclure que les approches futures seront en mesure de combler cette lacune et de faire évoluer les outils et les méthodologies vers un niveau supérieur. Par conséquent, nous espérons avec ce travail, inspirer les recherches futures dans le domaine pour atteindre des techniques d'analyse binaire toujours plus sophistiquées et automatisées.

Contents

List of Publications

- A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, D. Balzarotti. "Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem." In NDSS. 2020.
- A. Mantovani, L. Compagna, Y. Shoshitaishvili, D. Balzarotti, "The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study", In AsiaCCS. 2022
- A. Mantovani, S. Aonzo, Y. Fratantonio, D. Balzarotti, "RE-Mind: a First Look Inside the Mind of a Reverse Engineer", In USENIX 2022

Chapter 1

Introduction

In software security, we refer to Binary Analysis as the activity that enables the extraction of some information from the code of a binary executable. Executable programs are described by file formats (e.g., PE, ELF, and Mach-O) that organize the data of the program in different segments and sections, some of which are dedicated to encapsulate the binary executable code itself (i.e., the machine code). Machine code is more difficult to analyze (both for humans and machines alike) than its original source-level counterpart, and this poses several challenges to any software analysis solution that needs to operate on binary programs.

Several research fields indeed need to carry out binary analysis as their inherent goal is to analyse different types of executables. For example, malware analysts use it to identify the malicious behaviors that a sample can perform. In the area of vulnerability discovery, researchers implement instead binary analysis approaches to identify interesting code locations that can lead to dangerous vulnerabilities. Moreover, binary analysis is a frequent practice also in the context of firmware analysis, where researchers try to comprehend the security properties of a binary blob by looking at its internals. Since it represents a crossroad that different lines of research encounter for the development of novel analysis approaches, an extensive amount of research has been performed over the years on binary analysis solutions.

A common way to categorize these techniques is to split them into *static* and *dynamic* analyses. The former aims to extract information by looking at the code without executing it, while the latter collects the needed information by observing its behavior at run-time. Different techniques and tools fall into one of these two major families. For instance, dynamic code instrumentation (e.g., Frida [fri, DGHH+15]) approaches inject additional code into an application to monitor and trace it once it is executed. Thus, this methodology belongs to the dynamic category, as it requires the code to run. Similarly, debuggers such as GDB [gdb] are common dynamic tools that are routinely used to perform a step-by-step execution of the target program to inspect its internal state in some points of interest (such as the invocation of a certain API or the registers' value inside a basic block). On the other hand, decompilation is a good example of static technique, as it lifts the assembly code to a higher level of abstraction, normally known as *pseudocode*. Other forms of static analysis try to recover meaningful representations of the binary by working with disassembled code. In this last case, a very common example is the control flow graph recovery, which reconstructs how the different basic blocks are connected to each other inside the binary and displays this information either in a graphical form or in a more compact representation suitable for further processing.

Despite the fact that binary analysis is frequently used in many disciplines and that a large variety of approaches exist, complex activities often require the support of human experts. In fact, while some cases exist where fully automated solutions have completely replaced humans in binary analysis tasks, most analysis pipelines make use of automated components to simplify the job of the analyst, who is still responsible for at least a part of the careful and tedious manual investigation. More specifically, we can think of many binary analysis pipelines as human-in-the-loop approaches, where the analyst can interact with the analysis pipeline in various ways. However, the fact that human analysts might be involved in a task does not imply that they always play the same role in all systems.

In our research, we identified different ways in which human experts and automated techniques contribute to the solution of security-related problems. According to the human role and position with respect to the automated components, we can encounter three distinct configurations.

In the first scenario, the help of an analyst is needed at the beginning of the pipeline, for instance, to direct an automated component towards the "interesting" portion of the binary code or to manually identify and model new cases, whose information is then used as part of autonomous expert systems. In the second scenario, the analyst contribution is in the middle of the analysis pipeline. In this case, human experts are typically needed to manipulate, enrich, or filter the output of some tools before it is fed to other components. Finally, in the third category, human experts are employed at the last step of the pipeline, where they are often called to interpret the 'meaning' of the results of the autonomous system. For instance, while a machine can flag a new suspicious behavior in an unknown application, humans are still needed to decide whether such behavior is malicious in nature or is instead acceptable for the target program.

By looking at different examples of these three aforementioned cases, we selected three existing research fields that often require human intervention at different parts of their pipelines: malware analysis, vulnerability discovery, and reverse engineering. In the first case, we looked at the problem of identifying new types of packers, where human knowledge is needed to design custom rules to detect in-memory transformation patterns. As an example of human-in-the-middle configuration, we explored the domain of vulnerability discovery and we proposed a novel approach to identify potential memory corruption bugs in binary executables. Finally, we focused on the problem of binary reverse engineering, showing how reversers can 'understand' the behavior of a previously unknown binary.

These three contributions advance the state of the art in binary analysis and highlight that even state-of-the-art approaches are far from completely replacing human analysts in their job. On the one hand, this can lead to new tools and techniques that are able to mimic the human behavior for a specific task. For instance, in our work, we noticed that humans are primarily responsible for the comprehension of the code, whereas machines are still unable to accomplish this type of task. On the other hand, this thesis suggests that future research should make an effort to simplify the daily life of binary analysts in terms of design, usability and interaction.

1.1 Contributions

This thesis makes three separate contributions to the binary analysis field. We choose to investigate these three aspects, as they emphasize the different roles of human experts in binary analysis tasks.

Human as the entry point of the pipeline

A common binary analysis application in the context of malware analysis is malware packing. In this context, an open research problem on malware analysis is how to *statically* distinguish between packed and non-packed executables. This has an impact on antivirus (AV) software and malware analysis systems, which may need to apply different heuristics or resort to more costly code emulation solutions to deal with the presence of potential packing routines. It can also affect the results of many research studies in which the authors adopt algorithms that are specifically designed for packed or non-packed binaries. Therefore, a wrong answer to the question *"is this* *executable packed?"* can make the difference between malware evasion and detection.

It has long been known that packing and entropy are strongly correlated, often leading to the wrong assumption that a low entropy score implies that an executable is NOT packed. Exceptions to this rule exist, but they have always been considered as one-off cases, with a negligible impact on any large-scale experiment.

In our first contribution, we measure the prevalence of malicious samples that try to evade the static AV checks by hiding their malicious code with a layer of low-entropy packing. The challenge here is that to defeat this layer and correctly classify each malicious file, we need to analyse each sample (or at least the part of the code that handles the unpacking routine) present in our dataset. More specifically, our binary analysis task consists of two major steps i) understand if a sample is packed and in this case ii) determine the adopted packing scheme. Automating this procedure is fundamental, as we need to scale our analysis over a large number of malware samples. Indeed, for our experiments, we constructed a dataset of 50K Windows malware, way beyond what human analysts can process, independently on their expertise level. However, the automated component alone cannot really understand what the interesting part of the code that we want to track is. Hence, the human role in this task is to manually sample and analyze representative cases present in the dataset in an iterative process, each time translating the detected schemes into rules that help the automated part to identify other samples that adopt the same technique. This process needs to be repeated until the vast majority of packed samples can be categorized.

To accomplish this binary analysis task, we developed a dynamic analysis pipeline that is based on a set of heuristics to determine the packing scheme used by a certain malicious file. This requires an initial phase where a human analyst studies a portion of the samples to understand their internal functioning. After that, she can develop some "rules" to point the dynamic analysis to the specific code locations that show that particular packing scheme. After the analysis pipeline is set up, we organize our experiments in two distinct processes. First, we uncover those samples that show unpacking behaviors by dynamically instrumenting the memory accesses, and then, for those samples that are identified as packed, we implement a classification algorithm that relies on the execution of some specific assembly instruction typically adopted for some cryptographic operations as well as the observation of some values contained in the related memory areas before and after the operations.

Interestingly, after applying our automated binary analysis solution to the experiment dataset, we uncovered that low-entropy packing schemes are widespread in the wild, accounting for a total of 31.5% of packed samples. Moreover, low-entropy packing is based on a set of different cryptographic operations such as `XOR` encryption, transposition, and encoding – for which we propose a corresponding taxonomy and classification. This work was published in a research paper entitled "Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem".

Human as the middle point of the pipeline

In our second contribution, we look at the possible use of decompilers in the context of vulnerability discovery.

Decompilers are tools designed to recover a high-level language representation (typically expressed in C code) from binaries. Over the past five years, decompilers have improved enormously in terms of both the readability of the produced pseudocode and the similarity of the recovered representation to the original source code. Albeit decompilers are routinely used by reverse engineers in different disciplines (e.g., to support vulnerability discovery or malware analysis), they are not yet adopted to produce input for source-code static analysis tools. In particular, source code and binary vulnerability discovery remain today two very different areas of research, despite the fact that decompilers could potentially bridge this gap and enable source-code analysis on binary files.

In this part of the thesis, we conducted a number of experiments on real-world vulnerabilities to evaluate how the differences between original and pseudocode impact the accuracy of static analysis tools after running these on both the source and the decompiled code. One of the main limitations that currently hinder the feasibility of this approach is that the output of modern decompilers cannot be directly "re-compiled" by using traditional compilers (such as `Clang` and `GCC`). Therefore, many static analysis frameworks that rely on compiler passes to perform their vulnerability analysis cannot function in an automated fashion on the raw pseudocode. In addition to this, even for tools that implement a fuzzy parsing logic to read the code, and that therefore would be suitable to analyse the decompiled code, some complicated syntactical expressions contained in the pseudocode result in a difficult evaluation, often leading to some analysis errors.

Thus, to address these two problems, we introduce a semi-automated binary analysis approach, where the code lifting part is performed by three state-of-the-art decompilers (IDA, Ghidra, and RetDec). After that, humans are requested to manually "fix" the pseudocode to make it digestible for the static software testing tools. In our study, this manual procedure is used both to make the decompiled code "re-compilable" as well as to simplify the problematic patterns present in the lifted code. While the first part is needed for the static analysers to work properly, the second phase can tell us which code excerpts are potentially problematic during the code analysis and can therefore be the focus of future work in decompilation.

Interestingly, for this binary analysis task, human feedback is needed to aid the analysers to properly analyse the decompiled code. Inherently this affects the automation of the solution, as part of the task must be accomplished by hands. However, this approach comes with the advantage of statically finding vulnerabilities when source code is not available, which is quite typical in some scenarios (such as firmware images).

Remarkably, our results show that in 71% of the cases, the same vulnerabilities can be detected by running the static analyzers on the decompiled code, even though for several cases, we observe a steep increment in the number of false positives. To understand the reasons behind these differences, we manually investigated all cases, and we identified a number of root causes that affected the ability of static tools to 'understand' the generated code.

This work was published in a paper entitled "The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study".

Human as the end point of the pipeline

For the third and last contribution of this thesis we turn our focus to reverse engineering. In this case, while automated tools can be used to lift binary code to a higher-level representation that is easier to read and understand, humans still play a crucial role to *understand* this output. For instance, a system can reconstruct the control-flow graph of a binary, or a decompiler can be used to automatically rebuild the approximate source syntax of an assembly routine. However, the decompiler itself cannot describe the semantics of the code and its role in the overall program. Even though state-of-the-art work in reverse engineering has shown incredible advances in recent years, this task remains to date primarily a human activity. Therefore, we believe that studying how humans approach this type of task can be of great interest for researchers who want to improve automated algorithms or for training binary analysis experts. However, while experts in many areas (ranging from chess players to computer programmers) have been studied by scientists to understand their mental models and capture what is special about their behavior, the "art" of understanding binary code and solving reverse engineering puzzles remains to date a black box.

For this last application, we present a measurement of the different strategies adopted by expert and beginner reverse engineers while approaching the analysis of x86 (dis)assembly code, a typical static reverse engineering task. We do that by performing an exploratory analysis of data collected over 16,325 minutes of reverse engineering activity of two unknown binaries from 72 participants with different experience levels: 39 novices and 33 experts.

For this set of experiments, we designed the two binaries to capture the code comprehension of the participants. Although they did not contain a high number of basic blocks, the binaries include many typical assembly constructs that reverse engineers normally meet when approaching a static reverse engineering task. For example, one of the two binaries implements a simple server, whereas the second challenge consists of a list management application. The participants can reason on the binaries by using a tool (accessible as a web application) that performs some basic binary analysis (e.g., disassembly, control flow graph, and call graph recovery) and displays the results in a human-friendly fashion that mimics the interface of popular commercial tools. These problems require a manual approach, as the proper solution requires to fully understand the correct functionalities of the binary that lead to the target program point.

Our effort to *reverse engineer the behaviors of a reverse engineer* helped us to confirm some previously-proposed reverse engineering behaviors and disprove others. We also isolated several interesting features that we hope will be further analyzed by future research in this fascinating area. We explore this subject in the research paper “RE-Mind: a First Look Inside the Mind of a Reverse Engineer”.

1.2 Thesis Outline

The thesis is organized according to the following layout. Chapter 2 illustrates the background and concepts needed to understand the contributions of the thesis.

Chapter 3 is based on the paper “Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem” presented at the Network and Distributed Systems Security Symposium (NDSS 2020), and shows a first possible configuration of the human-in-the-loop binary analysis approach in the context of malware packing.

Chapter 4 presents the paper “The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study” accepted at AsiaCCS 2022, and illustrates our vulnerability discovery approach that allows to run static software testing tools on the recovered pseudocode thanks to a human-in-the-loop approach.

Chapter 5 is based on the paper "RE-Mind: a First Look Inside the Mind of a Reverse Engineer" accepted at the Usenix symposium 2022 and introduces our human study about static reverse engineering where the analyst represents the end point of the analysis infrastructure.

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

Chapter 2

Background

In this Chapter, we introduce the background concepts that are needed to fully understand the technical contributions provided in the follow up of the thesis. Since the topics span very different fields, ranging from malware analysis to vulnerability discovery, in each case we focus the discussion mainly on the notions that are related to the area of binary analysis and leave a more detailed cover of the related work to the individual chapters.

2.1 Malware Analysis

Malware analysis is the discipline that studies how to analyze malicious files to determine their nature and behavior. Thus, malware analysts need to develop a detailed understanding of the internals of the target operating system and architecture. For the experiments reported in this thesis, we focused on Windows malware analysis targeting the x86 architecture, and here we will provide a quick overview of the related basic concepts.

First of all, malware, as other forms of generic software, is structured according to a specific binary file format that depends on the host operating system. In our case, the reference file format is the Portable Executable (PE) format. The PE format organizes the information needed for the Windows OS to spawn the corresponding process and execute the wrapped code. It is made of different headers and sections whose main goal is to instruct the loader and the dynamic linker on how to map the file into memory. For instance, PE executables typically contain a code section (often named `.text`) that is mapped in memory with execute/read privileges, as well as a section for global variables (often named `.data`) mapped as read/write. Among the other sections, a special role is played by the `IAT` (Import Address Table), which contains the list of functions that are imported from external libraries, along with their addresses.

A second aspect that plays an important role in our work is the fact that malware often employs a variety of techniques to hinder the analysis and to hide the actual malicious payload. For instance, *packing* allows the malware author to decrypt some memory areas dynamically, generate valid code, and redirect the execution to these unpacked locations. Other techniques, such as *obfuscation* and *anti-disassembly*, do not hide the code but try instead to make it more difficult to understand (e.g., by flattening the control flow graph) or more difficult to analyze (e.g., by detecting the presence of analysis tools). On their side, malware analysts can also employ a wide range of techniques to analyze the target application and overcome evasive tricks. These techniques can be broadly divided into two categories, those based on static analysis and those on dynamic analysis. We will now provide more details about static/dynamic analysis approaches as well as an overview of packing, which plays a central role in understanding one of the contributions of the thesis.

2.1.1 Dynamic techniques

Dynamic analysis techniques for malware analysis are based on the idea of instrumenting the code to implement a certain monitoring logic. In this section, we introduce some basic concepts, especially w.r.t. the technologies that are commonly used in this discipline. For a complete view about state-of-the-art approaches, we point the reader to two more comprehensive surveys that illustrate the details that surround this important topic [ESKK08, OMNER19].

First of all, dynamic analysis techniques require running an executable inside a dedicated environment. Thus, the first design choice to implement this family of methodologies consists of selecting a *run-time environment* that can be an emulator, a virtual machine, or even a bare-metal machine specifically adopted for analysis purposes (thus set up so that the malicious binary cannot damage any actual user data).

The following step instead is related to *how* we want to instrument and monitor the process. Depending on the type of instrumentation, we can think about several approaches sharing the common aspect that the malicious sample is executed within an isolated environment referred to as the *guest*.

A first possible way to separate the existing dynamic analysis techniques is to classify them into *in-guest* and *out-of-guest*. With in-guest approaches, the actual analysis happens inside the guest machine, and this opens to two further implementation choices. Indeed, one possibility is to inject the monitoring logic at the userspace level directly into the process we want to track once this has been spawned. To achieve this, the tracer can modify some portions of the tracee's memory depending on the necessity. An example of this is *API hooking* that implements a function detouring mechanism to redirect the execution flow towards some monitoring logic before invoking the actual API [BB10, Fat04]. Another well-known approach is *dynamic binary instrumentation* that comes with the great advantage of a lower granularity level, as implemented by frameworks such as Frida [fri] and Intel Pin [RSCC04, LCM+05]. The overall idea is that the DBI framework takes care of duplicating some portions of the code of the tracee into a different mapped area that lives in the virtual address space of the tracee itself. Then, the duplicated code is augmented with the custom instrumentation, and the execution of the process interleaves between the original and the instrumented code, depending on what the analyst wants to observe. Userspace instrumentation is a good resource as it can collect fine-grained information about the executed code but comes with the disadvantage that malware authors can easily recognize when their malware is being analyzed and evade the detection mechanisms.

The counterpart of the in-guest userspace approaches instead tries to detect malicious actions from the kernel level. Kernel level monitoring has some main advantages, such as the fact that the analysis can reach more control over all the exhibited behaviors and performance-wise represents a better alternative compared to the user level strategies. In addition, this analysis paradigm can cope with the several malicious samples in-wild attempting to gain privileges inside the infected machine by performing actions directly at the kernel level. An example of this threat is the case of the so-called *rootkits* that modify the internal functioning of the operating system to conceal themselves (for instance hiding processes, etc.). On the other hand, the main disadvantage is the level of the granularity, e.g., we cannot reason at the instruction level. Kernel-side instrumentation typically consists of a kernel module installed inside the guest that monitors the execution of the system. When executing the module code (thus in kernel mode), the analysis can access all normal userspace processes, filter their system calls, monitor other aspects such as the creation/termination of a process and rely on the internal data structures of the operating system. Over the past years, this has become a standard technique, and indeed several modern antivirus companies deploy and distribute a kernel-level monitoring system.

A totally different approach is the one that is carried out by tools such as PANDA [DGHH+15] and DECAF [DQQY19, HPY+14] and falls under the category of the *out-of-guest* approaches. Indeed, for these cases, the instrumentation is injected during the emulation process, i.e., the malicious sample executes inside a system-level emulator (the guest) that internally translates the emulated ISA. The analysis works at the level of the emulated code, thus allowing for fine-grained measurements as it can see all instructions performed by the process. With this last approach, we do not affect the virtual address space of the tracee even though we pay in performances as this dynamic translation is very expensive.

Finally, the ensemble of run-time environment and instrumentation approaches represent the two design choices that are fundamental for the creation of a *sandbox*. According to the necessity, a sandbox could be designed to implement one or another of the previously described concepts. For instance, academic sandboxes often need to collect fine-grained information about a particular malware aspect. Thus for this type of sandbox, we could prefer to build an emulator-based machine and deploy an out-of-guest approach to monitor certain behaviors. Many industrial sandboxes instead have to handle thousands of files per day and therefore, they mostly care about lightweight techniques that do not introduce too much slowdown while executing the tracked process.

2.1.2 Static techniques

Static analysis techniques focus on extracting a set of parameters and features from the PE file to infer some aspects of the malware under observation. Two main approaches exist in this context.

A first direction is to perform code analysis in a static way, thus without actually executing the code but implementing some well-known program analysis techniques [SOA18]. The fact that the source code is not available inherently makes the development of such approaches more challenging, and in addition to this, several anti-analysis techniques can make the recovery of the code particularly challenging (e.g., packing).

A classical analysis paradigm is to infer some properties by studying binary representations such as Control Flow Graph (CFG) and Data Dependency Graph (DDG). This can give us information about the execution flow of the executable according to the value of some memory locations. For instance, we can use CFG and DDG to inspect some interesting function call sites and observe what the parameters of the invoked procedure are. Another program analysis technique used in the context of malware is symbolic execution, which can help to evaluate all execution flows of a binary executable that satisfy a certain constraint[1].

[1]We consider symbolic analysis as a static technique because its execution does not

On the other hand, a set of distinct solutions exist that work at the entire file level. The idea is that we can find some indicators by looking at the PE file format that suggests if a sample can be malicious or benign.

A first possibility is represented by those solutions that extract features processed by a Machine Learning (ML) classifier [UAB19]. In recent years, this has become a standard practice because it simply needs to fetch some features from the analyzed files, encode them into a features vector and pass it through a previously trained model. Static features can include different fields of the PE such as the name and the permission of the sections, the IAT, the timestamp, and other fields of this file format. More modern approaches take into account more elaborated features, for instance, the ones extracted from the binary code, such as n-grams of bytes or disassembled instructions.

At the other side of the spectrum, we find signature-based approaches that consist of matching sequences of bytes or specific patterns that identify a certain malware family [detay, peiay] or a certain packing or obfuscation technique. W.r.t. this approach the YARA rules [yar] represent the de-facto standard to implement expressive queries that can match such byte sequences.

In both cases, a static analysis technique has the pro that it is extremely fast to execute for each sample to analyze. However, they have the cons of the potentially many false positives that can also arise due to the anti-analysis techniques implemented by the malware authors.

A more comprehensive presentation of the related work about signatures is given in Chapter 3.5 whereas Chapter 3.6 presents a survey of state-of-the-art approaches in the context of machine learning for malware analysis, with an emphasis on packing.

2.1.3 Packing

Over the years, anti-analysis techniques have evolved in several directions, which include code obfuscation, compression, encryption, polymorphism, metamorphism, and runtime packing. This process and the real-world adoption of these techniques have been largely discussed in [YZA08, OSM11, RM13, UPBSB15]. Since packing, as used by today's malware, does not have a precise definition, it is essential to clarify which techniques we cover, and which we do not, in the rest of the thesis.

affect directly the system state (i.e., a symbolically executed program typically modifies symbolic memory/registers which are not the physical ones). However, note that this specific technique is actually in the middle between dynamic and static techniques and thus we acknowledge that other ways to categorize it exist

First of all, to draw a line between packing and other forms of anti-analysis, we consider packing only when I) the original code of the application is already present in the file but is NOT present in an executable form (i.e., it is encrypted, compressed, or otherwise transformed), and II) the original instructions are later recovered and executed at runtime. We consider instead obfuscation when the code is present in the binary and it retains the ability to be executed, even if it is hard to understand (for humans and/or automated tools) or analyze because it was re-written with the goal of hindering binary analysis. For the same reason, if a program encrypts all its data but not its instructions, we do not consider that as a form of packing in our study.

Dynamically-generated code (that also includes self-modifying code) is a generic term that refers to techniques used to generate or modify code at runtime dynamically. In a broad sense, packing relies on these techniques, and it is, therefore, a form of dynamically-generated code. However, not all forms of dynamically generated code are packing – for instance in the case of just-in-time compilers. To distinguish among the two, in our study we measure the size of the unpacked code and use this information to separate the cases when the actual application code is unpacked from the cases when just a small snippet of code (e.g., a shellcode) is generated at runtime.

Second, we limit our analysis to *runtime* packers that recover and execute the original code at runtime. Droppers that download a compressed archive from the Internet, unpack them on disk and then run the contained application are outside our scope (as both the dropper and the dropped files could be independently statically analyzed).

Finally, we do not consider emulators (like those included in the Themida packer) that transform the original instructions into a new instruction set and then execute them by using a custom emulator. In fact, in this case, the original code is *never* recovered, but instead permanently replaced with an (often randomized) instruction set.

2.2 Vulnerability Discovery

The goal of vulnerability discovery is to analyze different types of software to identify vulnerable patterns that can compromise the system that runs such programs. Since a very large and diverse set of approaches exist in this field, we propose two main orthogonal classifications while we cite the more comprehensive survey by Liu et al. [LSCL12] for more details.

The first way is to distinguish the cases where the source code is available from those ones in which the program source is not accessible, and the program is only available in binary form. This comes with the main implication that some critical information is lost at compile-time – such as the types, the data structures, and the variables/functions names. As an example of how this aspect influences the development of new bug-finding approaches, we can consider *fuzzing* [MHH+19].

Fuzzing is a popular technique that consists of repeating the execution of the program under test while mutating the input at each run. To monitor the coverage of the target code, fuzzers usually deploy an instrumentation that allows them to log when the execution encounters new program points (such as new basic blocks or new edges) [FMEH20]. When the source code is available, fuzzers normally inject the instrumentation needed at compile-time, for instance, by implementing custom LLVM passes [LA04]. However, when the program is accessible only in a binary form, researchers have to find alternative mechanisms to instrument the application. One way to solve this challenge is to emulate the target binary and inject the instrumentation during the phase of dynamic code generation (as done by popular state-of-the-art fuzzers like AFL++ [FMEH20]). Another technique instead is binary re-writing, which instruments the code directly at the assembly level [NNTH+21, DBXP20]. In all cases, fuzzing applied to binary-only software requires a completely different approach compared to the source code scenario. Similarly, this concept holds for other vulnerability detection techniques such as *symbolic execution*, where previous work demonstrates that the presence of the source code enables the symbolic executor to reach better performances [PF20].

A second way to look at the vulnerability discovery techniques is to distinguish between the cases where we execute the target and the cases where we do not. This translates into two opposite approaches, i.e., *static* and *dynamic* analysis. Fuzzing clearly falls into the second category, as it consists of repeated executions of the target. Static software testing instead is representative for the first category as it performs a series of analyses (typically at the source code level) that investigate some program representations to detect vulnerable patterns [LC10]. For instance, static analyzers can query the Abstract Syntax Tree (AST) to take into account the basic semantics of the program being evaluated. More advanced static software testing tools instead are able to construct also the Control Flow Graph (CFG) and the Data Dependency Graph (DDG) to develop data tracking techniques that are useful to identify, for instance, the code locations that certain user input can reach, thus determining if a certain statement is safe or not (for instance a buffer indexing). Since a chapter of this thesis focuses on static software testing, we will present the related work on these aspects later in Section 4.1.

2.3 Reverse Engineering approaches and tools

Reverse Engineering is a broad topic that covers several different activities. Therefore, in this section, we emphasize what aspects we studied in our work and the actual focus of the thesis.

In system security, we refer to binary Reverse Engineering (RE) as the activity by which a human, the Reverse Engineer, analyzes an executable file, either in whole or in part, to recover design and implementation information useful to understand the program functionalities. This implies that the reverse engineer has to interact with the low level mechanisms of an architecture and the way a certain operating system manages these aspects. Overall, for our studies we focused on the x86 architecture running both Linux and Windows operating systems.

Depending on the context (e.g., malware analysis, vulnerability discovery, firmware analysis), the output of a reverse engineering analysis can be different. However in all cases the analyst is interested in reconstructing the logic of the program and in understanding which conditions must be met to reach a specific location in the code — which can be related to a bug or to a suspicious behavior in the case of malicious files [Yur13].

Independently from its goal, the RE process usually involves different phases, and different tools are used to inspect the program and collect the required information. Some popular frameworks that support the analyst in this complicated task are interactive disassemblers, such as IDA Pro [idaay] and Ghidra [ghiay]. These tools combine multiple functionalities (e.g., a disassembler, a decompiler, a debugger) in an interconnected and interactive user interface, which allows the analyst to inspect an enriched representation of the binary code.

2.3.1 Disassemblers

Disassemblers are popular computer programs that lift the machine code into assembly language. We can refer to the recovered assembly code as *disassembly*, as it represents a close, but not perfect, reconstruction of the original assembly code. In fact, some information is lost when the assembler generates the machine code. For instance, embedding data in the code areas is a source of errors for disassemblers as they might try to parse the data bytes as if they were instructions, often leading to an incorrect output.

Other reasons that can cause errors are indirect branches, functions without an explicit `CALL` site, position independent code, and hand-written assembly.

Nevertheless, two main algorithms exist to implement a disassembler and try to address some of these points. The first one is *linear sweep*. This naive approach starts decoding instructions from the first byte and continues until it reaches the end of the code section or an illegal instruction. A more advanced approach is *recursive traversal*, that relies instead on a careful control flow analysis. More specifically, it begins with the entry point and then follows and visits each branch instruction either in a depth-first or breadth-first fashion.

Finally, state-of-the-art disassemblers improve the recovered representation to make the disassembly reading easier for the analysts. For instance they resolve libraries function calls, replacing their addresses with API symbols and often reporting their parameters. Moreover, they compute some metadata that can be used by the reverse engineer to understand the internals of the application, such as the Xrefs to and from a certain function, i.e., the callers and the callees of the currently visited function.

2.3.2 Decompilers

In the current section, we present the basic concepts behind the design of a decompiler while in Section 4.1 we will present the state-of-the-art work related to this topic.

Decompilers are tools designed to recover a high-level C-like representation of the assembly code. Typically, we refer to this high-level representation as *pseudocode* or simply *decompiled code*. In recent years, using the pseudocode to reverse complicated functions have become a standard technique, as it allows to automatically reconstruct C language constructs such as loops, variable assignments, and function calls. However, compilation is by definition an irreversible process, as a considerable amount of information is stripped away when generating the final executable. For instance, the type systems information and the size of many stack buffers are lost and difficult to reconstruct.

To try to recover some of the list information, the first step that the majority of modern decompilers adopt is to lift the disassembled code into an intermediate representation (IR). Hence, a set of passes is executed on the IR to reconstruct the expressions (expression propagation), the uses of variables (data flow analysis), and the type system (type analysis and type propagation).

In the last phase, the structuring module is in charge of transforming the elaborated IR into high level constructs such as if/else/while statements.

After that, the decompiler backend can emit the resulting pseudocode. As a consequence of all these modifications and lifting steps, the decompiled code can contain complicated expressions and statements. A very frequent example of this is the frequent use of `GOTO` statements, which are used to reconstruct complex control flow topologies. Another case is the use fields in a C struct. Since decompilers cannot precisely recover struct definitions, they often mis-represent them as arrays, in which fields appears like elements withing the array itself.

Chapter 3

Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem

This first chapter focuses on a classical malware analysis problem known as packing. In the following sections, we will show how we implemented a dynamic analysis approach to perform fine-grained measurements about the packing methods implemented by the several malicious samples we collected in our dataset. Moreover this study exemplifies our first scenario of human intervention, where the analyst has the role to guide the automated component towards the unpacking code and thus works as the entry point of the pipeline.

Both benign and malicious applications have valid reasons to hide or disguise their internal behavior; the former to deter attempts to reverse engineer their code and break software protection mechanisms, and the latter to evade detection from antivirus engines and security products. A wide range of *anti-reversing* techniques exist that modify the binary code of a program to make it difficult for humans to understand and for computers to analyze. Among them, *code obfuscation* and *runtime packing* are the most frequently adopted by both malware and goodware authors.

On the one hand, *Obfuscation* aims at rewriting a program in a way that preserves its semantic but complicates its form. This can be done, for example, by flattening the control-flow, inserting dead code or opaque predicates, or by adding sequences of instructions that can confuse disassemblers ([LK09, BM08, BCCO16, MC06, CTL98, SRX14]). Obfuscation plugins are often included in popular compiler toolchain infrastructures (e.g., Obfuscator-LLVM [JRWM15] and Proguard [Laf04]).

On the other hand, *Runtime Packing* is a technique that was originally introduced to save disk space by compressing (at rest) and decompressing (at runtime) the code of an application. More generally, the term is used today to describe a class of techniques designed to store a compressed, encrypted, or otherwise encoded copy of the original program – thus preventing any static analysis of the code itself. Packed samples rely on a short unpacking routine that allows them to reconstruct the original application code in memory and then execute it.

While the exact fraction of packed malware samples is still unclear, in a recent study by Rahbarinia et al. [RBP17], the authors found that 58% of the malicious downloaded files are packed with an off-the-shelf packer. However, their estimation does not take into account the presence of custom packers (35% of packed malware adopts custom packers, according to [MP10]). Moreover, the authors rely on signature-based tools that are known to generate many false positives – as we show in more detail in Section 3.5. In any case, the widespread adoption of packing makes the problem of correctly and efficiently answering the question *"is an executable packed?"* fundamental in malware analysis. In fact, many classes of techniques – such as static analysis, clustering, and similarity among samples – do not work or provide poor results in the presence of packed executables. This forces researchers to pre-process packed samples by introducing a very costly and time-consuming dynamic unpacking phase, or by completely replacing static approaches with more resilient solutions based on dynamic analysis.

A wrong classification of packed samples can also pollute the datasets used in many malware analysis studies. For instance, researchers often rely on datasets that include both packed and not packed samples, and errors in this separation can lead to unreliable or difficult to reproduce experimental results.

To solve these problems, the security community developed a number of efficient tests to assess the presence of packing. Historically, the Shannon entropy of a program was adopted for this purpose, as both encrypted and compressed data are characterized by a very high entropy — which can be easily distinguished from that of machine code. While early studies (e.g., [LH07]) classified executables just according to their *average* entropy, researchers quickly moved towards entropy computations performed at a lower granularity, i.e., by relying on sliding windows or by calculating the entropy of individual sections. These more fine-grained techniques were often described as very successful in identifying the presence of packing. For instance, Han and Lee [HL09] reported 99% of accuracy and precision by looking at the entropy of individual sections. Another well-known approach to identify the presence of packing relies on the use of custom signatures, as applied by popular tools like Detect It Easy [detay], Manalyze [manay], and PEiD [peiay]. However, this solution is prone to errors, and it is unable to identify previously-unknown packing routines – as we show in more detail in Section 3.5.

Since entropy became a discerning metric to discover packed code, both researchers and malware authors experimented with techniques to pack executables while maintaining the entropy low. For instance, in 2010, Baig et al. [BZRL12] discussed the possibility of using different encodings to reduce the entropy and evade the checks performed by antivirus software. However, the study was purely theoretical, and the authors did not provide any evidence of the actual adoption of such schemes in the wild. Two years later, Ugarte et al. [UPSS+12] found that samples belonging to the *Zeus* family contained trivial countermeasures to tamper with entropy checks. In this case, the malware authors padded the encrypted data by inserting the same byte (or a subset of bytes) repeated multiple times – in Section 3.3 we inspect the details of how this and the other schemes affects the entropy. Since then, this phenomenon has been sporadically mentioned by malware analysts, but it has never been discussed in detail, and its adoption by malware authors has never been measured. Therefore, even if the existence of low-entropy packing was known to researchers, it was often dismissed as statistically irrelevant and with a negligible impact on practical experiments. As a result, researchers (such as in [LH07, PLL08b, SUPS+11, UPSGF+14, RBP17, UPBSB15]) continued to resort to entropy-based metrics and static signatures to identify the presence of packing. For instance, in the extensive analysis and large-scale measurement of malware packing performed to date [UPBSB15], the authors selected their samples from VirusTotal [viray] by querying for files with an entropy greater than seven.

Research questions

Even though security experts do not rely solely on entropy to identify packed samples, there is no systematic study that measured how prevalent low-entropy packing schemes are in the wild and whether existing techniques are able to correctly classify these samples. Moreover, there are popular tools (discussed in Section 3.5), academic papers (discussed in Section 3.6), and even books ([SH12, LAHR10]) that still adopt the approximation *packed* $\approx$ *high entropy*. The goal of this chapter is to show that this simple approximation is not correct in a large number of cases, and to improve our knowledge of low-entropy packing by answering the following research questions:

1. Which tricks and which packing techniques are used by real-world malware to lower their entropy?

2. How widespread are these techniques in the wild? Is low entropy packing a significant trend that needs to be considered when designing malware experiments?

To answer these questions, we assembled a dataset containing 50,000 low-entropy malicious samples belonging to multiple families. Our methodology to analyse them consists of a human-in-the-loop approach where the expert guides a dynamic analysis tool that, thanks to the initial input of the analyst, can classify each sample and categorize the scheme and transformations applied to the packed code. Our analysis pipeline reported that over 30% of them adopt some form of runtime packing.

For this reason, we decided to investigate if other features can still be used to detect the presence of packing. In fact, while some papers (e.g., [LH07, PLL08b, UPBSB15]) and tools (e.g., [peiay, detay]) consider only entropy to distinguish packed from non-packed malware, state of the art solutions use a combination of different static features, often based on PE structural properties. Therefore, we introduced two additional research questions:

3. Are existing *static* solutions able to distinguish low-entropy packing from unpacked samples?

4. If not, can we do that by combining all static features that have been proposed to date in related works, or new research is needed to solve this problem?

In Section 3.5 we show how the most popular and actively maintained static tools available today perform on our dataset. Finally, in Section 3.6, we collected all the static features that have been proposed in previous studies as reliable indicators of the presence of packing. We then trained several classifiers on the union of these features and tested them on our dataset of low-entropy malware (containing both packed and not packed samples). It is important to note that our goal was not to design a new classification scheme based on the combination of all existing features but only to understand whether these features can successfully classify samples in the presence of low-entropy packers.

3.1 Background

3.1.1 Entropy of Executable Files

Entropy is a metric to measure the uncertainty in a series of numbers (or bytes) or, in other words, to capture how difficult it is to independently predict each number in the series. The difficulty in predicting successive values can increase or decrease depending on the amount of information the predictor has about the function that generated the numbers, and any information it retained about the prior numbers in the series.

In particular, the Shannon entropy H of a discrete random event x tries to predict the number of bits required to encode a piece of data, as given by the formula:

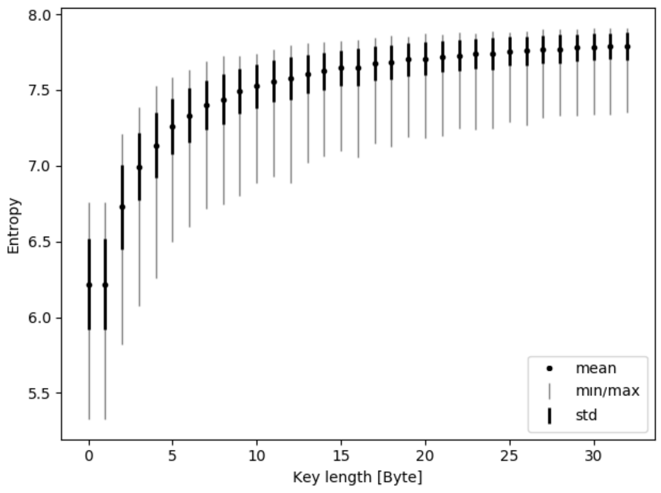
$$\mathrm{H}(x) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_2(\mathrm{P}(x_i))$$

where $\mathrm{P}(x_i)$ is the probability of the i^{th} unit of information (such as a number) in event x's series of n symbols. This formula generates entropy scores between 0.0 and 8.0 when considering that each symbol can have 256 values as it is the case for binary data. Both *lossless compression* and *encryption* functions typically generate high entropy data. In fact, lossless compression functions start by generating a statistical model for the input data, then use such a model to map input data to bit sequences in a way that frequently encountered data will produce a shorter output than infrequent ones; this removes predictability, which increases the entropy. The same applies to encryption functions, as they are specifically designed to generate unpredictable data.

Since the generation of a *packed executable* often relies on compression and/or encryption to disguise the application code, packed files are usually characterized by having a high entropy. As a consequence, entropy was the primary metric used in the past to classify *packed* executables [LH07].

However, many file formats for executables, such as Portable Executable (PE), Executable and Linkable Format (ELF), and Mach Object (MO) divide the file into a number of isolated sections. Obviously, this way of partitioning an executable affects the distribution of its entropy. For instance, machine instructions are often redundant, thus resulting in middle-range (typically 5-to-7) entropy scores, while strings of English text result in even lower entropy values (on average 4.7 [For07]) due to the limited

Figure 3.1: `.text` section entropy w.r.t. XOR encryption



number of characters they employ. To better discriminate among different areas of an executable, researchers replaced file-level entropy scores with a more fine-grained computation performed at the level of individual sections [HL09] or by applying a (sliding) windows over the program's bytes [PLL08a, UPSGF$^+$14].

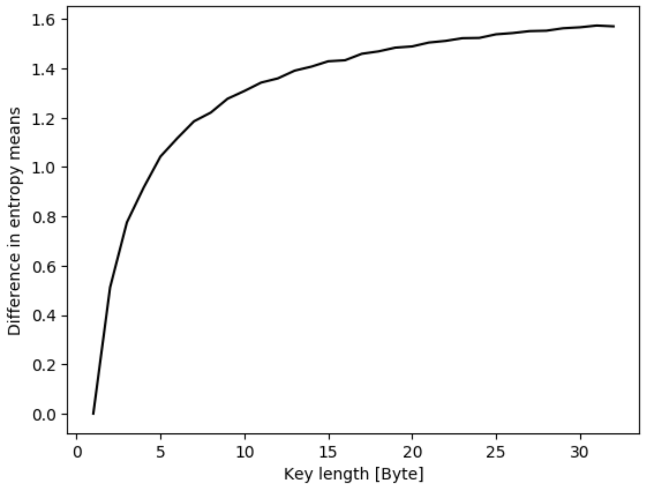
3.1.2 Entropy and XOR Encryption

Since packing usually encrypts code to hide it, we set up an empirical experiment focused on PE x86 code encryption to distinguish between average and high entropy values of plain and encrypted code. On Windows 7, we installed the top 10 applications from the Microsoft Store [Mic], including top browsers and the Visual Studio IDE. We then randomly selected 1,000 PE executable files, both 64 and 32 bit, from the Program File folder[1]. For each of them, we calculated the entropy of their `.text` sections; then we *XORed* the `.text` section with a randomly generated key, and we re-calculated the entropy of this new encrypted data. We repeated the experiment 128 times, changing the key length from 1 to 32 Bytes.

Figure 3.1 shows the evolution of the entropy for different key lengths. The circle shows the mean of the 128 experiments, the thick vertical line is

[1]We ensured that each file was not previously packed by using the tool we presented in Section 3.2.

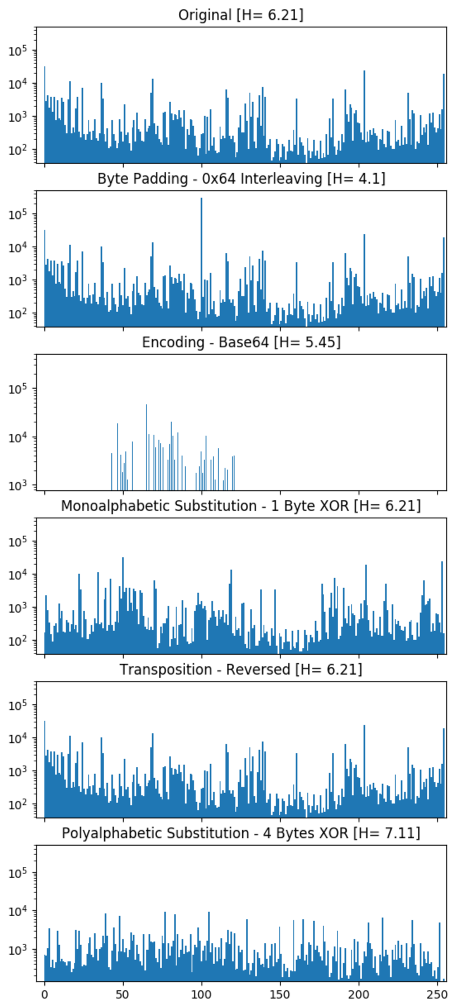
Figure 3.2: Encrypted `.text` section – difference in entropy means



the standard deviation, and the thin vertical line shows the range between the maximum and minimum value. As shown in the image, the entropy slowly grows accordingly to the length of the key. When the key length is only 1 Byte long, the entropy does not change as this is just a substitution of the plain-text code and does not alter the frequency of the symbols. Our test shows that the average entropy of real-world plain x86 code is around 6.2 ± 0.3, and by using a 2-bytes key the entropy increases to 6.7 ± 0.3. Figure 3.2 shows the difference between the means of the entropy of the XORed code and the original code – emphasizing the rapid effect that the key size has on the entropy of the data.

Finally, we observed that state-of-the-art approaches [UPBSB15] and frequently used tools (e.g., [detay, manay, pefay] – discussed in Section 3.5) adopt 7.0 as entropy threshold to separate packed and not packed executables. According to our graphs, this value is obtained on average by xor-ing the code with a key of 3 bytes. In the rest of our thesis, we will use this threshold to distinguish *low entropy* data ($H < 7.0$) from *high entropy* data ($H \geq 7.0$) and we use this value to construct our low-entropy malware dataset.

Figure 3.3: Byte Frequency Distribution w.r.t. Schemes



3.2 Prevalence of Low-Entropy Packing

3.2.1 Dataset

We built our dataset by downloading 50,000 Portable Executable (PE) files, (excluding libraries and `.Net` applications), randomly selected among those submitted to VirusTotal [viray] between 2013 and 2019. We only selected PE samples classified as malicious by more than 20 antivirus engines, and such that the entropy of each section, of the entire file, and of potential overlay data[2] were less than 7.0 (as motivated in Section 3.1.2). We adopted these conservative criteria to ensure that a sample is certainly malicious and contains neither compressed nor encrypted data.

Furthermore, we collected a second smaller dataset containing 476 samples used in APT campaigns [aptay], which satisfies the same low entropy constraints. From now on, we will refer to this dataset as the *APT dataset*. The samples belonging to the APT dataset were collected over a period spanning from 2015 to 2018.

3.2.2 Analysis

To carry out our first experiment, we designed and implemented **Packer Detector** (hereafter, *PD*), a dynamic analysis tool built on top of the PANDA [DGHH+15] analysis framework. The goal of *PD* is to precisely trace unpacking behavior by monitoring when a sample executes a memory area which it previously modified. The analysis of a single sample produces as output several text files that are subsequently analyzed to reconstruct the behavior of the sample.

Since the original code of a packed sample is hidden, the unpacking procedure must carry out some operations to retrieve, restore, and store somewhere in memory the unpacked code. For this reason, *PD* dynamically executes the target sample on a virtual machine (Windows 7, 64 bit) by spawning the corresponding process and monitoring its registers and memory content. Each sample is executed until the main process exits or until a maximum timeout of 40 minutes is reached. The virtual machine gateway points to an INetSim[3] instance, which provides fake HTTP/S and DNS responses to deceive the sample under analysis into believing that it is connected to the Internet. Despite INetSim delivers fake files based on the file extension in the HTTP request (e.g., .html or .exe), it is configured to avoid returning any executable code, since we do not want to analyze malware which uses external data because it can violate our low entropy constraints.

PD collects write accesses on the memory of the main process and, if applicable, of its child processes, and stores this information as a list [WL] (Writes List) of tuples. The memory accesses are detected thanks to the PANDA callback `PANDA_CB_VIRT_MEM_AFTER_WRITE` which is raised every time a process performs a memory write. If the PID matches with that one

[2]The overlay is just appended data to the end of the executable file that is ignored when loading an executable into memory because it is not covered by the PE header. Anyway, opening the executable file in reading mode will allow access to the entire file including the overlay portion.

[3]https://www.inetsim.org/

of the process under analysis (or with one of his child ones), we store a tuple in the list $[WL]$. Each tuple contains the Program Counter (PC) register pointing to the instruction that triggered the write operation and the target address AW (Address Written) of the operation. This means that for the i-th write operation to the address AW_y performed by the instruction at address PC_x, we have a tuple $\langle PC_x, AW_y\rangle_i \in [WL]$. If the write operation involves more than one byte, the system stores them separately. For instance, if a sample executes the instruction " `mov WORD PTR [0x1000], 0x4142` " at the address `0x1234`, *PD* manages the size directive adding the tuples $[\langle \texttt{0x1234}, \texttt{0x1000}\rangle, \langle \texttt{0x1234}, \texttt{0x1001}\rangle] \in [WL]$. Furthermore, if the PC reaches a previously written address in the tuple $\langle PC_x, AW_y\rangle \in [WL]$ (i.e., AW_y points to instruction that it is getting executed), it copies the tuple into another list $[WXL]$ (Written and eXecuted List); at the end of the execution this list will contain all the written-then-executed addresses and the PC values that triggered the write operation. When the sample terminates, *PD* analyzes the $[WXL]$ list: if the sample is packed, the list is not empty and it encompasses some memory regions of consecutive addresses (modulo the x86 length of instructions) that contained the unpacked code. However, when a sample manually loads a Dynamically Linked Library (DLL) and then executes one of its functions, *PD* would detect this behavior as part of an unpacking routine. To remove this noise, our tool further checks whether the program counter points to code that belongs to a DLL, and remove these cases from our analysis. We also use a threshold of 800 bytes on the length of the $[WXL]$ list to exclude samples which simply decrypt a short shellcode, a behavior that we do not consider a form of packing and that anyway would likely not significantly affect the overall entropy.

The heuristic adopted by *PD* can also generate false negatives (i.e., packed samples detected as not packed) if the sample runs incorrectly because of an unexpected crash, incorrect command-line arguments, missing dependencies, or virtual environment evasion [4]. To avoid the risk of polluting our dataset with wrong labels, we decided to conservatively discard the samples that did not exhibit a sufficient amount of runtime behavior, and that therefore might have been incorrectly executed. This includes samples that did not invoke at least ten disk- or network-related syscalls as well as samples whose executed instructions did not span at least five memory pages. For this reason the *PD* hooks the disk/network-related syscalls (for instance

[4]Virtual environment evasions are techniques aimed at detecting whether an executable is running on bare-metal or a virtual machine (regardless of it being emulated or based on a hypervisor).

NtOpenFile or *NtCreateFile*) relying on the syscall hooking interface offered by PANDA. We also keep track of the code coverage of the sample, i.e., the number of instructions executed compared to the total number of instructions in the executable sections (typically `.text`). The goal of these selection criteria is not to detect evasive malware, which is still an open problem, but to remove from our dataset those samples that could be incorrectly classified as not packed simply because they failed to run. By applying these simple heuristics, we removed a total of $3,705$ malware samples from our dataset. Based on our conservative thresholds, it is safe to assume that the remaining samples executed long enough to at least unpack their code. For this reason, from now on, we consider $46,295$ as the total number of samples over which we compute our results.

3.2.3 Results

During our analysis, we run into a class of samples that, while packed with a high-entropy scheme, evaded our set of filters described in Section 3.2.1. These samples contained encrypted data, but the data was not stored in any of the section nor the overlay area. For instance, a family of file infectors adopted this technique to inject its encrypted code in an area created between the PE header and the first section. While this data belongs neither to the PE header nor to any section, it is automatically loaded in the main memory at runtime (unlike, for instance, the overlay data that needs to be manually loaded by the program). Moreover, since the size of this encrypted code is small with respect to the size of the entire file (approximately 2.6%), it has little impact on the total entropy of the file. In addition to the area between the PE header and the first section, we have also discovered samples that used the empty area (if present) among sections to store their packed data. In total, 11.6% ($5,386/46,295$) of the samples in our dataset adopted this interesting, and to the best of our knowledge previously undocumented, scheme to store packed code in a way that evades common entropy-based checks. Among them, the two prevailing families were *hematite* (64%) and *hworld* (35%). Since these samples successfully evaded our entropy checks but without using a low-entropy scheme, we decided to consider them as a separate category in our dataset.

Over the remaining low-entropy samples, our tool discovered that a stunning 31.5% ($14,583/46,295$) employed some form of packing. This shows that entropy alone is a very poor metric to select packed samples and that roughly one-third of the samples with entropy lower than seven are still adopting some form of runtime packing to prevent static analysis. This per-

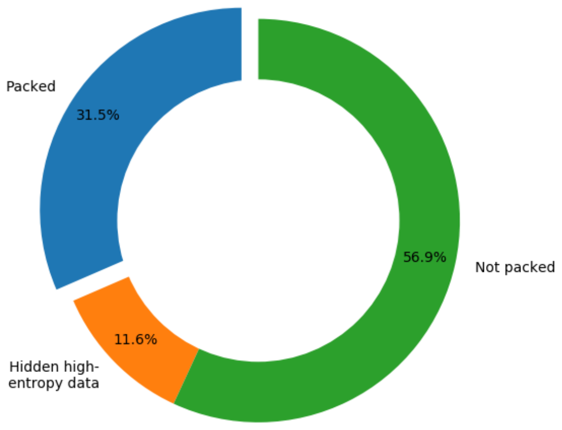


Figure 3.4: Dataset composition (cardinality = 46, 295)

centage is even higher if we exclude the samples with hidden high-entropy data. In other words, if we pick a random malware sample that contains no information with entropy higher than seven, according to our experiments there is a 35.6% probability (14, 583/40, 909) that it is packed with a low-entropy scheme. The overall composition of our dataset is summarized in Figure 3.4.

We also downloaded the VirusTotal report of every sample in our dataset and using *AVclass* [SRKC16], a malware labeling tool, and we have identified the family associated to each sample. Table 3.1 reports a ranking of the top ten families in the packed and not-packed categories.

Finally, in the *APT dataset* we did not find any sample that has hidden high-entropy data, while low entropy packing schemes were adopted by 15% of the samples. This shows that low entropy schemes are a well-known practice for malware authors nowadays and the phenomenon is significantly widespread in the wild, leading us to our next research question: *which packing techniques do malware authors adapt to keep the entropy below the suspicious threshold?*

3.3 Low Entropy Packing Schemes

In this section, we describe the experiments we conducted to enumerate and analyze the different techniques adopted by malware authors to keep the

Table 3.1: Top 10 families distribution in our dataset

Packed	**%**	**Not packed**	**%**
sivis	28.0	lamer	10.4
unruy	11.6	daws	8.8
vobfus	9.5	vbclone	8.0
dealply	5.4	sivis	7.5
upatre	4.1	triusor	4.5
shipup	4.0	flystudio	4.0
gepys	3.5	zegost	3.9
vilsel	2.9	mailru	3.6
sality	2.3	high	2.9
hematite	2.0	nitol	2.4

Table 3.2: Low Entropy Scheme

Scheme	**Effect on Entropy**
Padding	Decrease
Encoding	Decrease
Mono-alphabetic Substitution	Unchanged
Transposition	Unchanged
Poly-alphabetic Substitution	Slightly increase

entropy below detectable levels, and measure the frequency in which they appear in our dataset. We emphasize that we refer to *low entropy packing schemes* regardless of their effect on the entropy (increasing, decreasing, or unchanged), as long as such schemes produce *low* entropy data according to our results in Section 3.1.2. Moreover, it is important to note that sophisticated packers often involve several layers of unpacking routines, in which the first layer unpacks the second one, which in turn unpacks the next layer and so on until the original code is reconstructed. However, for our purpose, we only need to study the first unpacking layer, as it is the only visible from a static analysis point of view and the only one that determines the entropy of the data. As we will discuss later in this section, malware authors may also decide to use stronger encryption in deeper layers as long as they keep the entropy of the first layer low.

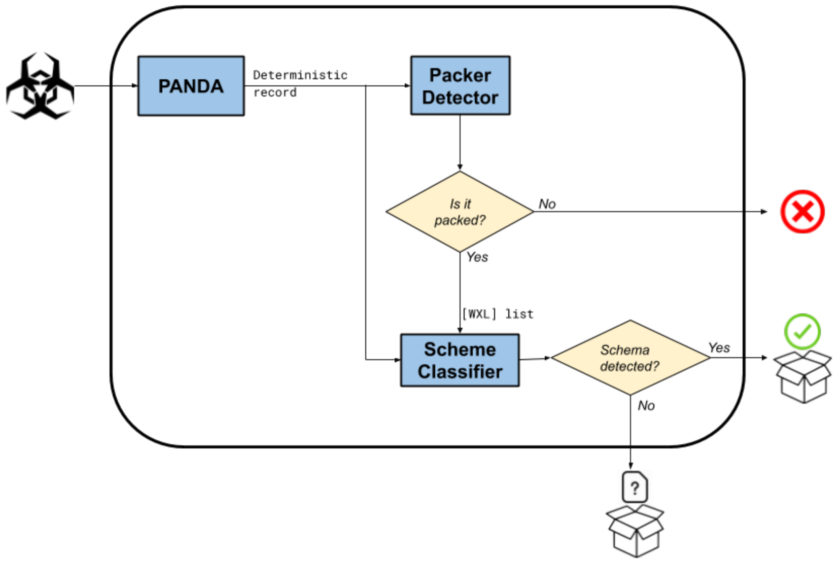


Figure 3.5: Architecture of our analysis tools

3.3.1 Schemes Taxonomy

We can divide the low entropy schemes observed in the wild into five main categories, summarized in Table 3.2. The table also shows the effect that each scheme has on the final entropy. While some techniques can be used to effectively lower the entropy of data (and therefore 'hide' an already packed sequence of bytes), others can only maintain (or slightly increase) the current entropy, thus requiring to be applied as standalone solutions on the original application code.

Byte Padding includes all techniques in which additional low-entropy data is added to the packed section to decrease the overall entropy. This data typically consists of a single byte, or a repetitive subset of bytes, that are either appended at the end of the code or interleaved with the packed instructions. The unpacking routine, accordingly, skips over the padding while restoring the original instructions. *Byte padding* alone is not a packing technique, and therefore it is often used in combination with other encryption or compression schemes.

Encoding-based schemes decrease the overall entropy by representing the packed information using a different number of bits, thus encoding the same data with a different alphabet of symbols. Although we observed some samples applying well-known encoding schemes to pack their code, other malicious samples often implement their **custom encoding** (during our analysis we just observed 6-bit alphabets). As encoding-based schemes can lower the entropy of high-entropy data, they can be used to mask multi-layers approaches that also employ traditional encryption packing.

Monoalphabetic Substitution-based approaches aim at replacing every single byte in the packed payload with a different byte, computed either by using a simple algorithm (e.g., a *XOR* with a 1-byte key) or by looking up each symbol in a translation table.

Transposition is another technique that does not alter the byte distribution and the entropy of the data. In this case, either individual bytes or sequences of bytes are shuffled around to recompose the original code. Sometimes the transposition scheme is fixed, while in other cases the samples embed the 'instructions' to reassemble the bytes in the correct order in the packed data itself.

Polyalphabetic Substitution schemes are simple cryptographic techniques that extend simple byte substitution by using multiple substitution alphabets. Common examples of this approach are the classic Vigenère cipher or the `XOR` encryption with a multi-byte key. While these techniques usually result in an increased entropy score, the use of very short keys (e.g., 2-4 bytes, as shown in Section 3.1.2) do not significantly modify the byte distribution, and therefore it limits the increase of the entropy level.

3.3.2 Schemes in action

To give an idea of how such schemes work on a real example, we have taken a benign file from the samples we used in the experiment in Section 3.1.2. In particular, we have chosen a sample with the entropy of its `.text` section corresponding to the average entropy we previously measured in the same experiment. Then, we applied an example of each of the different low-entropy scheme listed above on its `.text` section. As shown in Figure 3.3, we implemented respectively: padding – interleaving the byte `0x64` after each original byte (thus doubling the size), encoding – base64, substitution – XOR with a one-byte key, transposition – byte ordering reversed, poly-alphabetic substitution – XOR with 4 Byte long key. The graph shows for each byte $[0, 255]$ (represented on the x-axis) its frequency in the data (on the y-axis) plotted on a logarithmic scale. In the padding plot, the `0x64` byte is the most frequent; this scheme is noteworthy for the way that it effectively decreases the entropy, with the downside of increasing the original source size. The encoding plot contains only the bytes belonging to the *base64* scheme, decreasing the entropy accordingly. Looking closely at the Substitution plot, the reader can notice that the frequencies are shuffled w.r.t. the original distribution; for example, given that we used the byte `0x32` as the key, the original `0x00` byte frequency has been moved ($\texttt{0x32} \oplus \texttt{0x00} = \texttt{0x32}$) to the `0x32` (50 in decimal) frequency. Given that the frequency distribution does not consider the order, the transposition and the original plot are identical, including the entropy. Lastly, the poly-alphabetic substitution is characterized by a more uniformly distributed bytes frequency, and in fact it is the only one that increases the entropy over the 7 threshold.

3.3.3 Scheme Classifier

Once Packer Detector identifies a sample as using some form of runtime packing, a more refined analysis is needed to detect to which of the previously introduced five categories the low-entropy packing scheme belongs to. To accomplish this second set of experiments, we developed another dynamic analysis tool, also based on PANDA, that we call the **Scheme Classifier**. This tool relies on the output of Packer Detector and applies some heuristics based on the fact that every packing scheme needs to follow the same steps: i) locate and access the source buffer that contains the low entropy packed data, ii) perform operations on such data, iii) write the unpacked data in the destination buffer. We sketched the architecture of our tools and how they are integrated together in Figure 3.5.

It is worth remembering that the output of Packer Detector is a list of tuples, named $[WXL]$. Each tuple $\langle PC_x, AWX_y\rangle \in [WXL]$, contains the program counter PC_x of the instruction that triggered the write operation in memory, and the target address AWX_y where the information was subsequently stored and executed. Accordingly, this information defines the memory regions that contain the destination buffer of the unpacking routine. Moreover, given that PANDA supports the deterministic record and replay of a sample, the tool performs its analysis by replaying the same trace that was recorded by Packer Detector. For each unpacking operation, the Scheme Classifier disassembles (using Capstone [capay]) and analyses the assembly instructions executed just before the memory write to the destination buffer. It then parses the assembly code and, by relying on the PANDA framework, it reads the values contained in the registers and in the referred memory addresses. The instructions and the values we obtain are used for two reasons: first, the Scheme Classifier performs a backward data-flow analysis to locate the source buffer (S_b) – where the packed data is located. This corresponds to the identification of all the memory read operations which are supposed to contain the packed code which is being unpacked. For this purpose the Scheme Classifier relies on the PANDA callback `PANDA_CB_VIRT_MEM_AFTER_READ`, which is triggered every time a memory read operation is performed by the process we are tracking (in our case the sample under observation). Second, the Scheme Classifier extracts all the mathematical operations that are applied to the source bytes to generate those in the destination buffer (D_b). To achieve this second step, the tool makes use of the `PANDA_CB_INSN_EXEC` exported by PANDA, which allows us to analyse all the instructions actually executed by the sample. Since we already know the program counter values corresponding to interesting memory writes (as Packer Detector provides such values in the [WXL] list as mentioned before), we just need to track these specific values and the previous mathematical instructions without taking care of the other instructions executed. It then uses these two pieces of information to classify the possible packing scheme adopted by a sample, by following this sequence of rules:

1. The Scheme Classifier first applies some rules to identify the presence of known encoding schemes (*base64*, *base32*, ...) in the source buffer S_b. If it recognizes a standard encoding, the Scheme Classifier marks the sample as *encoding.*

2. If the frequency of the bytes in the source and destination buffers is the same, but bytes appear in a different order, it classifies the scheme as a *transposition.*

3. If the byte distribution in the D_b is shuffled with respect to the S_b and the entropy is the same, the Scheme Classifier reports it as *mono-alphabetic* substitution.

4. It then looks for arithmetic/logic operations (XOR, ADD, ...) that modify the S_b and write to the D_b. If it finds an interesting cryptographic operation, it tries to extract the potential encryption key by analysing the disassembly and reading the value stored in the registers and memory. For example, if before a memory write, the target value was previously XORed with a 2-bytes fixed value, this means that the sample is using a XOR encryption with a 2-bytes key. In this case, the unpacking scheme is classified as *poly-alphabetic* (the *mono-alphabetic* case is captured in the previous step).

5. If no interesting operations are detected, the Scheme Classifier looks at the entropy of the input buffer. If it is the same that would be

obtained by applying a known encoding to the output buffer, but the set of symbols is different, it marks it as a potential *custom encoding*.

6. When S_b and D_b match except for a subset of bytes that is present with high frequency in S_b and with low frequency in D_b, the Scheme Classifier infers that *byte padding* is being used.

7. When the Scheme Classifier cannot apply any of the previous techniques, it marks the scheme as *unknown*, and leave it for a further manual investigation.

3.3.4 Results

In Table 3.3 we report the result of the Scheme Classifier; namely, the distribution of low-entropy schemes that we observed in our fine-grained analysis performed over all the 14, 583 samples found by Packer Detector. When possible, we also specify the specific type of transformation that is employed. It is also worth noting that the heuristics applied by the Scheme Classifier are extremely time-consuming: in average, they require around 90 minutes per sample.

Table 3.3: Scheme distribution

Scheme	**Type**	**%**
Padding	-	8.0
Encoding	standard	3.9
	custom	0.5
Mono-alphabetic Substitution	XOR	29.8
	ADD	5.2
	ROL/ROR	0.5
Transposition	-	0.3
Poly-alphabetic Substitution	XOR	46.9
	ADD	2.8
Unknown	-	2.1

`XOR`-based encryption is by far the most prevalent technique in our dataset, accounting for more than 76.7% of the analyzed samples. It is present both in its simplest form (`xor` with a single constant byte) in 29.8% of the cases, as well as with multi-byte keys of various length. A basic (`base64`) encoding was used in 3.9% of the samples, while padding accounted for slightly more than 8%. In 97.9% (14, 276/14, 583) of the cases the Scheme Classifier detected a tangible unpacking scheme, so we are reasonably sure that the vast majority of the samples discovered by the *PD* are actually packed. The remaining 2.1% (307/14, 583) contains either samples adopting unforeseen schemes that we could detect with our tool, or possibly samples using other forms of dynamically-generated code that were not removed by our heuristics.

3.4 Human Role

Despite the fact that our approach achieves a proper scalability to cope with the high number of malicious files in our dataset, the human still plays a fundamental role in the overall view of the analysis pipeline. Indeed, the pipeline still represents a human-in-the-loop approach, where the human provides with some input to redirect the analysis towards specific code locations. This source of input is linked to the code understanding skills of the analyst and it is due to the fact that state-of-the-art frameworks cannot really comprehend the different portions of the code. For instance, in our case, before writing our tools based on PANDA [DGHH+15], we needed to manually analyse some samples to understand how the unpacking phase works and how the schemes operate to generate a low entropy cipher text. This can require a non negligible amount of time because of the many difficulties that arise from the study of malicious code. In many cases in facts, we had to face custom packers whose internal functioning was completely unknown, and that required a careful job of decoding of all procedures involved in the actual unpacking part.

Only after manually analysing a representative portion of the samples we could automate the whole BA approach. This requires a second human activity as the analyst has to indicate a set of "rules" that capture the interesting behavior and encode such behaviors according to the expressiveness of the language adopted. In our case, the "rules" are written in form of C++ plugin for the binary analysis framework PANDA [DGHH+15]. Other tools and approaches can generalize this concept or expose different APIs and different langauges. However, writing down the automation step is the outcome of a code comprehension phase that tries to point the adopted tool to investigate a certain portion of the code.

3.5 Signature and Rule-based Packer Detection

So far, we have discussed the nature and measured the prevalence of different low-entropy packing schemes adopted by real malware in the wild. Our experiments show that this is a ubiquitous phenomenon and that entropy alone cannot be used as a reliable indicator to identify the presence of packing. However, beyond simple entropy, security researchers also proposed other tools and techniques to identify packed samples. In this second part of the thesis, we measure to which extent these alternative approaches allow us to distinguish packed from non-packed samples in presence of low-entropy schemes.

Signature-based solutions identify known packers by relying on a (typically manually curated) set of patterns that are associated with known off-the-shelf packers. Existing engines for pattern detection vary in complexity, from the ones that work on raw bytes to those that recognize and reason about the file structure. This difference consequently influences the expressive power of the employed signatures.

For our experiments, we have chosen the most popular and actively maintained tools available today that rely on open signatures: Detect It Easy [detay], Manalyze [manay], and PEiD [peiay].

Detect It Easy (DIE) adopts an open architecture of signatures, based on a scripting language similar to JavaScript. This language provides great flexibility and expressive power that allows DIE to declare complex and fine-grained signatures.

PEiD is another widely used tool for statically analyze PE files, looking for most common packers, cryptors, and compilers. PEiD signatures only contain low-level byte patterns, which can be optionally matched either at the PE file's entry point or anywhere in the file.

Manalyze is a static analyzer for PE files, composed of several plugins. Its *packer detection* plugin adopts signatures based on the name of the PE sections (for example the UPX packer compresses all existing sections and renames them as `UPX0`, `UPX1`, etc.) as well as several rule-based heuristics designed to capture anomalies in the PE structure typically associated with the presence of packing, including unusual section names, sections both writable and executable, low number of imported functions, resources bigger than the file itself, and sections with entropy greater than 7.0 – that is the same threshold we used for constructing our dataset.

DIE and PEiD also have a dedicated component for the entropy. Even if they have different thresholds (DIE 7.0 by default, PEiD is not open source so we cannot report the precise number), all of them classify an executable as packed when its entropy is greater than a certain value. Also, the python module *pefile* [pefay], often used to parse and edit PE headers, contains a function that estimates if the input executable is packed, and it is solely based on the entropy. This fact highlights how this metric is still relevant nowadays and how popular tools still support the correlation between high entropy and packing.

Signature scan results

Probably because of its finely tuned signatures, `DIE` detects no well-known packer in our entire dataset. This is not a bad result, as we expect the vast majority of samples in our dataset to rely on custom packing routines. In fact, popular off-the-shelf packers are widely known and easily recognizable, thus making it more unlikely for them to 'fly under the radar', which is the main advantage of adopting a custom low-entropy scheme.

In contrast, both `PEiD` and `Manalyze` generated a large number of alerts, as summarized in Table 3.4. The result of both tools are comparable, but also quite surprising, as they consistently detected the presence of packing more often in not packed samples than in the packed group. For instance, signature-based mechanisms recognized 1.7%-to-2.6% of samples in the packed group but misclassified 9.6%-to-13.1% of the entries in the not packed dataset. For `Manalyze` this is due to the presence of sections names that correspond to those used by some off-the-shelf packers. We cannot say for sure why the malware authors used those names. They could be fake clues used on purpose to deceive automated tools into believing that a sample is packed with a known packer and, consequently, to trigger the use of unpacking routines that would invariably fail on the program.

Table 3.5 shows the top five common packers detected by these signature-based systems. Given that our dataset only contains samples with low entropy, the presence of compressor packers (UPX, UPolyX[5], and ASPack) and a crypto packer (Petite) immediately suggests that these are probably all false positives. In any case, we run existing unpacking tools for UPX, UPolyX, and ASPack and confirmed that all of them failed and found no sign of packing. We also manually inspected samples reported as PolyEnE and Petite (as no tools are available for these packers) and again confirmed that there were no traces of these packers. A closer look at the matching PEiD signatures revealed that they were often too general, or designed to match anti-disassembly tricks and strings that could also be used in other contexts. The only case we were able to confirm consisted of three samples recognized as packed (also confirmed by our Packer Detector) with Beria. Samples packed with Beria contains two types of byte, which we call "original" and "metadata". During the unpacking routine, the metadata bytes

[5]UPolyX is basically a scrambler (thus, it does not affect entropy) that needs a UPX packed input file to produce a number of different output files.

are evaluated through an algorithm that computes the correct offset where the original bytes need to be written inside the destination buffer. This approach does not increase the entropy as the original bytes appear unchanged (just not in the correct order) and the metadata bytes follow a strict and repetitive pattern.

Table 3.4 also reports the alert generated by the `Manalyze` heuristic component, which flagged 57% of the packed samples and 23% of non-packed samples as likely packed. By investigating the internal logs, these misclassifications are mainly due to the presence of unusual section names or of executable permission on writable sections.

Table 3.4: Signature-based detection results. “sig.” stands for signature and “heur.” stands for “heuristics”

Dataset	Manalyze sig.	Manalyze heur.	PEiD	sig $\wedge$ PEiD
Packed	242 (1.7%)	8,358 (57.3%)	386 (2.6%)	214 (1.5%)
Not Packed	2,518 (9.6%)	6,023 (22.9%)	3,438 (13.1%)	2,487 (9.4%)
Hidden H-E	0 (0%)	14 (0.3%)	2 (0.1%)	0 (0%)

Table 3.5: Well-Known Packers Detections

Packed		Not Packed	
Name	**%**	**Name**	**%**
UPX	1.1	UPX	10.0
ASPack	0.5	ASPack	1.2
UPolyX	0.5	UPolyX	1.2
Petite	0.1	PolyEnE	0.7
PolyEnE	0.1	Petite	0.4

In conclusion, existing signature-based tools are well suited to detect the presence of common off-the-shelf packers, but unfortunately, generate a large number of false alerts on non-packed samples. Even worse, these false positives are more frequent on non-packed malware than on those packed by using low-entropy schemes, which suggest that these samples are difficult to classify statically.

3.6 ML-based packing detection

If the use of signatures or hard-coded heuristics failed to detect the packed samples in our dataset, this does not rule out the possibility to find other

Table 3.6: Results of ML experiments

Alg.	**Train**	$Err_{notPack}(W)$	$Err_{pack}(W)$	$Err_{notPack}(\tilde{W})$	$Err_{pack}(\tilde{W})$
SVM	75%	4.43%	25.01%	4.12%	24.57%
	50%	4.31%	28.41%	3.97%	26.20%
	25%	4.44%	32.01%	4.11%	29.85%
MLP	75%	6.34%	12.70%	5.86%	12.15%
	50%	6.87%	16.14%	6.24%	14.73%
	25%	6.89%	11.91%	6.33%	12.93%
RF	75%	0.20%	32.77%	0.23%	31.54%
	50%	0.18%	29.46%	0.20%	28.46%
	25%	0.21%	28.84%	0.20%	26.83%

discriminatory features that can help identify even the most elusive form of packing.

Therefore, in the following section we explore alternative static analysis approaches proposed by other researchers. We first survey the state of the art and gather all the features that have been proposed in the past. In order to evaluate the performance of these features, we implement a machine learning classifier based on the union of all these features. With this, we do not intend to propose a new classification system, nor to compare existing approaches with respect to each other. Instead, like in previous sections, our goal is simply to evaluate whether these features are able to correctly classify our low-entropy set of samples.

3.6.1 Feature Extraction

Lyda and Hamrock [LH07] were the first to take into consideration entropy (computed initially over the entire file) as a metric to classify packed malware. The basic idea was then refined to calculate the entropy for each section of the sample [HL09] or over small byte windows [UPSS+12].

Researchers also investigated the use of machine learning to train a classifier over a large number of static features. To start with, many authors [CkKOcR08, PLL08b, STF09, TZ09, DN12, STMF09] proposed features that captured specific anomalies that packers introduce in the PE file format. Even if such features could identify off-the-shelf packers with a high level of accuracy, samples making use of custom packers could successfully evade all the checks based on the file structure alone. As a consequence of this, malware researchers extended the sets of features to include some that would allow capturing the presence of custom packing routines better. For instance, the approaches proposed in [PLL08a], [ASPE13], [SUPS+11] and [UPSGF+14] adopt a larger collection of features that include structural PE attributes, heuristic values, entropy-based metrics, byte n-grams, and disassembly opcodes. Other heuristics includes the raw data per virtual size ratio (computed over all the sections), the ratio of the sections with virtual size higher than raw data or the fact that the entry point is outside any section.

In 2010, Jacob et al. [JCN+12] extended the idea of n-grams analysis by proposing a methodology for detecting the similarity between packed samples, which takes into consideration the n-grams fetched from the code sections and tunes them according to the noise introduced by the different packers. In 2019 Lim et al. [LRK+19] proposed to analyze executable files as a stream of bytes, and discuss several statistical properties to determine the randomness level of each stream. Finally, we considered some dynamic unpackers ([CMF+18, SYS+08]) that rely on static features to understand if the unpacking procedure is correctly terminated. Unfortunately, while some of these works included custom packers in their experiments, none of the aforementioned proposals considered the presence of samples packed with low-entropy schemes.

We are only aware of two exceptions to this rule. In 2012, Ugarte et al.[UPSS+12] performed several experiments which included some samples of the Zeus botnet, one of the first families that adopted a low entropy packing scheme. However, the approach proposed by the authors is tailored to the single specific case documented in their paper and would fail to address other common low-entropy techniques. Therefore, we did not include this technique in our study.

Raphel et al. [RV15], instead, focused their study on the use of XOR-based encoders. In this work, XOR encryption is recognized as a form of obfuscation mainly used to encrypt small parts of the code like shellcodes. The idea was to refine the use of entropy to recognize samples that adopted a XOR-based scheme. Mainly, their approach relies on 5 steps: (i) extraction of fragments from files; (ii) computation of entropy for each fragment; (iii) concatenation of fragments; (iv) computation of entropy for each concatenated fragment; and (v) construction of a similarity distance matrix based on the previously computed values for each file pair in the dataset. Like in the previous case, this solution targets a very specific problem and is not directly applicable to the type of packers we are studying because the authors designed it with the purpose of detecting small portions of encrypted code (essentially schellcodes). Anyway, we considered this approach in our evaluation.

To summarize, we can group the proposed features in six different families:

- PE Structure: values extracted from the PE headers (and thus often

Table 3.7: GreyEnergy dynamically allocated memory regions

Name	Size [Byte]	Permissions
Mdst	0x20200	RW
Moffsets	0x808	RW
Mtmp1	0x200	RW
Mtmp2	0x200	RW
Mexe	0x24000	RWX

referred simply as PE).

- Heuristics: features produced as a result of common knowledge about characteristics of packed PE files.
- Opcodes: sequences of assembly instructions extracted from the executable sections.
- N-grams: sequences of N bytes extracted from the entire file or some of its sections.
- Statistical features: evaluation of statistical properties about the randomness of a sequence.
- Entropy features: features based on the computation of entropy with respect to some areas of the file (sections, overall file, sliding windows).

Table 3.8 summarizes all the presented static analysis approaches and lists the categories of features as well as how the authors constructed the dataset they used in their experiments.

3.6.2 Evaluation of Static Features on Low-entropy Packers

In this section, we evaluate the reliability of the previously discussed static analysis techniques in detecting packed samples.

To assess this, we use our dataset of 40,909 samples (i.e., all running programs minus the samples with hidden high-entropy data, because those can be detected with proper entropy analysis). For the same reason we also decided not to include in the dataset any *high-entropy* packed samples, i.e., those using traditional packing schemes such as UPX, ASPack, and Armadillo. In summary, our dataset contained 14,583 samples packed with *low-entropy schemes* and 26,326 *not packed* samples. From now on we will refer to this subset of samples as the *ML dataset*.

For each malware in the *ML dataset*, we extracted all the features adopted by the 15 state-of-the-art approaches discussed in the previous section, and summarized in Table 3.8. We refer each approach through an index i, where $0 <= i <= 14$. The i^{th} approach applies several ML algorithms using an input vector of n features $V(i) = [f_{0_i}, ..., f_{n_i}]$ where f_{m_i} represents the m^{th} feature of the $i - th$ approach, with $f_{m_i} \in \mathbb{R}$. To simplify the experimental setup, we joined the feature vectors $V(i)$, for $i = 0, .., 14$, in a single vector $W = [V(0)|...|V(14)]$. If two or more approaches rely on the same feature, we considered it only once. We point out that the vector W includes the entropy features as well. To verify if entropy still plays a role as discerning metric, we define the vector $\tilde{W}$ as the feature vector containing all the features of W except for all the entropy features.

We split the *ML dataset* into *train set* (TrS) and *test set* (TeS) and we run the classifiers on different subsets of TrS and TeS. TeS is composed by a subset TeS_{packed} of packed samples, and a complementary subset $TeS_{notPacked}$ of not packed samples, s.t. $TeS_{packed} \cup TeS_{notPacked} = TeS$.

We indicate FP and FN the sets of false positives and false negatives samples, respectively. The set FP contains the not packed samples which are classified as packed, while FN contains the packed samples which have been classified as not packed. In particular, we focus on the number of errors the classifiers make respectively on packed and not packed samples:

$$Err_{notPack} = \frac{|FP|}{|TeS_{notPacked}|} \tag{3.1}$$

$$Err_{pack} = \frac{|FN|}{|TeS_{packed}|} \tag{3.2}$$

We show our results in Table 3.6. For each classifier we report the ratio between *training* and *testing* sets, and the $Err_{notPack}$ and Err_{pack} obtained by using the two feature vectors W and $\tilde{W}$ (i.e., with and without entropy features). Our experiments, summarized in Table 3.6, indicate that none of the classifiers provide a high level of accuracy – with the best model implementing MLP and achieving the 11.91% as false negatives rate but also the 6.89% as false negatives rate. It is worth noting that in most of the cases, the classifiers show a high Err_{pack} ratio, which means that a significant number of packed binaries are classified as not packed. This suggests low entropy schemes can effectively be used by malware authors to bypass classifiers based on static features alone. For instance, we noticed that several files have PE headers appearing perfectly normal (sections named with standard names, entry point correctly located inside `.text`, a high number of entries in the IAT, etc.). While this somehow decreases the level of obfuscation provided by traditional packing schemes, it still succeed in protecting the application code against automated static analysis routines.

With this we do not want to say that static features used in previous studies are useless. In fact, they do much better than entropy alone. However, in presence of low-entropy packed samples all classifiers trained on these features perform quite poorly, and certainly far worse than what was reported in previous experiments. For instance the authors of [HL09] claim to reach the 0.0% as false positive rate and 2.5% as false negative rate by only relying over entropy metrics while in works that employ ML features, the authors declare to obtain a false positive rate of 0.8% ([SUPS+11]).

3.7 Case studies

In this section, we discuss in more details three malware samples that implement low entropy packing techniques. We also investigate why they are (or are not) detected by the features introduced in the previous section. We hope that this can help to understand better the internals of real-world low-entropy packing schemes and the reason why malware writers adopt them.

3.7.1 Case I: Simple XOR Encryption

For the first case study, we look at a sample[6] that belongs to the *berbew* family. By looking at the code located at the application's entry point, it is easy to identify a simple XOR encryption algorithm that applies a fixed 4-bytes key to decrypt in place the `.text` section. The hardcoded decryption key is `0x6d02676d`. Since the first and last digits are the same, the encryption only raises the overall entropy of the packed code to 6.9; it is reasonable to believe that this repetition was a conscious decision introduced to lower the entropy. The malware author also padded the code of its `.aciof` section with a large number of `0x90` bytes (corresponding to the `nop` x86 instruction) – likely for the same purpose.

Although this sample can evade any entropy-based check, it is easily detectable by using other static features. In fact, this PE file contains several anomalous values – including the `RWX` permissions of the `.text` section and the non-standard name of the section `.aciof`.

While this scheme is relatively simple and not particularly interesting from a research point of view, we decided to include it in our case studies because it is representative of the vast majority of low-entropy techniques

[6] `md5= 7186708dd7a1b0dbf9294909679ec30b`

we observed in our dataset and because our Scheme Classifier could automatically categorize it. Next, we are going to present two more complex cases that required manual investigation to be classified.

3.7.2 Case II: Transposition Scheme

Our second example is a sample[7] that belongs to the arsenal of a cyber-espionage group dubbed GreyEnergy [greay]. Since 2015, this malware has been used as part of attacks against energy companies and other high-value targets in Ukraine and Poland. Most specifically, the binary is a *loader*, i.e., the code in charge of stealthily loading the real malware into the target system.

The sample hides the packed data in the `.text` section, within the range `[0x1000, 0x211ff]`, for a total of `0x21200` bytes. This packed data, `PackedSrc` from now on, has an entropy of 6.59, and it contains, in a scattered disposition, all the data that is necessary to create a valid PE file. A simplified algorithm of the packing scheme is presented in Algorithm 1 (the original technique also involved operations between integers of different sizes that we omit for brevity). The unpacker uses five memory regions dynamically allocated (using the `VirtualAlloc` API) as reported in Table 3.7.

The first step is a call to `init(Moffsets, n)` to initialize the `Moffsets` memory region (line 1 and 6), that represents an array of integers. This array is initialized with `n` integers s.t. $Moffsets = \{\forall i = 0...n | 0 \leq Moffsets[i] < n\}$ and every number in range $[0, n]$ is contained in the `Moffsets`. Those properties allow the unpacker to later use the `Moffsets` region as a lookup table that implements a bijective function $f : [0, n] \rightarrow [0, n]$.

The algorithm then splits the `PackedSrc` and `Mdst` in 514 chunks of 256 bytes each and it copies every chunk from `PackedSrc` to `Mdst` (line 4), but not consecutively: it uses the `Moffsets` table (initialized in line 1) for looking up the proper offset in the destination buffer (line 3).

After that, it splits again `Mdst` in chunks (this time 257 chunks of 512 bytes) and each chunk is copied into `Mtmp1` (line 8); then, one byte at a time, it is copied into `Mtmp2` by using the offsets specified in the re-initialized (in line 6) `Moffsets` table (line 10). At the end, `Mtmp2` is directly copied into `Mdst` (lines 12-14) and in turn into the executable region `Mexe` (line 16).

When the unpacking procedure is completed, the sample parses the unpacked PE in the `Mexe` memory, and loads (using the library function `LoadLibraryA`) every `dll` requested in the *Import Table*. Then it modi-

[7] `md5= 7a7103a5fc1cf7c4b6eef1a6935554b7`

fies the *Process Environment Block* structure's `ImageBaseAddress` field [8], so that it points at the very beginning of the unpacked PE file. Finally, it jumps to the entry point of the unpacked PE[9].

The remarkable achievement of this scheme is that the byte distribution, and consequently the entropy, of the packed and unpacked regions are identical. Moreover, from the static analysis point of view, this sample is undetectable using both signatures and ML techniques, among the ones described in the previous sections.

Algorithm 1: GreyEnergy unpacking scheme

```
init (Moffsets, 0x202);
for (i = 0, j = 0; i < 0x202; i += 1, j += 0x100) do
    offset = Moffsets [i] * 0x100;
    memcpy (Mdst [offset], PackedSrc [j], 0x100);
end
init (Moffsets, 0x200);
for (i = 0; i < 0x8080; i += 0x80) do
    memcpy (Mtmp1, Mdst [i], 0x200);
    for (j = 0; j < 0x200; j += 1) do
        Mtmp2 [Moffsets [j]] = Mtmp1 [j];
    end
    for (k = 0; k < 0x200; k += 1) do
        Mdst [i+k] = Mtmp2 [k];
    end
end
memcpy (Mexe, Mdst, 0x20200);
```

3.7.3 Case III: Custom Encoding

Our final sample[10] uses two layers of packing. The second (deepest) layer, relies on a traditional XOR encryption scheme (with an 8-bytes key) and ROR/ROL loops that produced packed data with high entropy. To mask this fact, the malware authors added a first layer of packing that reduced the entropy from 7.63 to 6.57 by adopting a custom encoding scheme.

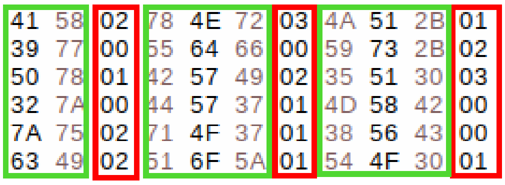
The first layer relies on the content of two sections: `.rsrc` and `.rdata`. Figure 3.6 shows some bytes extracted from the `.rsrc` section. It is clear that the data consists of sequences of three bytes (highlighted by the green

[8]The ImageBaseAddress field contains the address where the legitimate process executable is loaded.

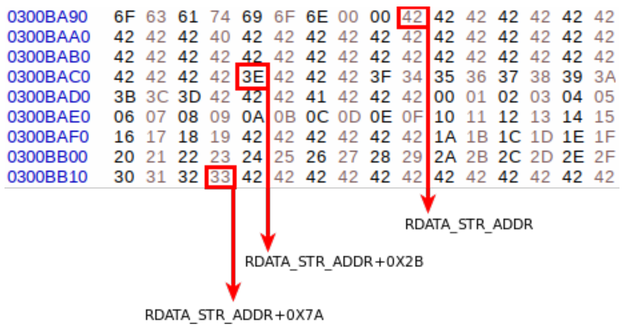
[9]`md5= ab8df9b7389ae890c3396a238bdc4606`

[10]`md5= c03bc642c5a49c55efb2d07a7272af2e`

Figure 3.6: Pattern stored inside the .rsrc section



```
41 58 02 78 4E 72 03 4A 51 2B 01
39 77 00 55 64 66 00 59 73 2B 02
50 78 01 42 57 49 02 35 51 30 03
32 7A 00 44 57 37 01 4D 58 42 00
7A 75 02 71 4F 37 01 38 56 43 00
63 49 02 51 6F 5A 01 54 4F 30 01
```



```
0300BA90   6F 63 61 74 69 6F 6E 00 00 42 42 42 42 42 42 42
0300BAA0   42 42 42 40 42 42 42 42 42 42 42 42 42 42 42 42
0300BAB0   42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
0300BAC0   42 42 42 42 3E 42 42 42 3F 34 35 36 37 38 39 3A
0300BAD0   3B 3C 3D 42 42 42 41 42 42 42 00 01 02 03 04 05
0300BAE0   06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
0300BAF0   16 17 18 19 42 42 42 42 42 42 1A 1B 1C 1D 1E 1F
0300BB00   20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
0300BB10   30 31 32 33 42 42 42 42 42 42 42 42 42 42 42 42
```

Figure 3.7: The string `0x0300ba99` in `.rdata` section

rectangles) within the range [`0x2b`, `0x7a`], separated by a single byte [`0x00`, `0x03`] (highlighted by the red rectangle in the image).

The `.rdata` section contains a buffer filled with some characters without a particular meaning (mainly the "*B*" character, `0x42` in hexadecimal). However, from the offset `0x2b` to `0x7a` (as shown in Figure 3.7), the buffer contains bytes ranging from `0x00` to `0x3f`.

Algorithm 2 summarizes the unpacking procedure in pseudo-code. The code loops through all values in the `.rsrc` Section (line 3) and uses each byte as offset to access the string (lines 4-5). If the value of the read byte is `0x42`, the algorithm moves to the next byte (line 6), while others are combined four at a time by adding each value to the previous one shifted by six bits (lines 7-8). The result is finally written to another memory region (line 12), before resetting the counter and restarting the loop (lines 13-14).

The PE structure of this file does not contain any anomaly, and the above-described custom scheme (that uses the same symbols of the Base64 scheme) is able to hide the packed code from n-grams and opcodes analysis; therefore this sample evades all the previously described ML techniques.

Algorithm 2: Pseudocode of the first layer

```
i = 0;
res = 0;
for $addr \leftarrow RSRC\_START$ to $RSRC\_END$ do
    offset = readByte(addr);
    byte = readByte(RDATA_STR + offset);
    if $byte\ != 0x42$ then
        res = res << 6;
        res = res + byte;
        i = i + 1;
    end
    if $i == 4$ then
        writeToMemory(res);
        res = 0;
        i = 0;
    end
    addr = addr + 1
end
```

3.8 Conclusions

In this work, we conducted a set of experiments on real-world malware to demonstrate that existing static approaches fail to take into consideration the threat represented by low-entropy packed malware and that this phenomenon is relevant enough that cannot be ignored when designing malware experiments. Although previous works [UPSS+12, RV15] have discussed the existence of low-entropy packing schemes as case studies, our work is the first to study this phenomenon in depth, and to measure the prevalence of this technique over a large dataset.

While it might be true that high-entropy file are often packed, our experiments show that the opposite is not correct – i.e., the fact that the entropy is low is not sufficient to conclude that the file is most likely not packed. This is important as many studies and tools still use the entropy alone to classify a sample as packed or not.

The results of our large scale dynamic analysis performed on 46,295 samples shows that 31.5% of low-entropy files were packed, proving that this type of malware represents an actual and widespread reality. As final proof of our results, we have also analyzed a reduced set of 476 APT-linked (Advanced Persistent Threat) malware that represent state of the art for complex attacks. We found that in this context the phenomenon of low-entropy packed malware occurs with a frequency of the 15%. In Section 3.3 we have categorized how such schemes keep their entropy low and the frequency in which this technique is adopted in the wild.

We then investigated why actual static analysis techniques are unable to detect the presence of low-entropy packing. We have studied two kinds of approaches: those based on signature/heuristic in Section 3.5, and those based on machine learning in Section 3.6. On the one hand, signatures are just well suited to detect the presence of common off-the-shelf packers, while heuristics generate a large number of false alerts on non-packed samples. On the other hand, we evaluated the performance of static feature-based classifiers, when entropy is no longer a reliable way to detect packers. Unfortunately, our experiments show that this is not the case as even the best classifier was able to detect only 70% of the packed samples in our dataset. Our results show that the accuracy of these classifiers degrades drastically in the presence of low entropy packers, which means that the results reported in the past relied significantly on the entropy to discern between packed and not packed files, and that the datasets employed may have not correctly represented the low-entropy packers that we found in the wild. Moreover, the machine learning experiments tell us that the static features proposed so far are inadequate and needs to be extended to allow for a accurate classification of packed samples. This does not mean that these approaches, including simple entropy-based measurements, must be abandoned. Instead, our work emphasizes the need for new solutions to this open problem, and that the existence of low-entropy packing must be considered in future experiments conducted by researchers and practitioners.

Moreover, we share[11] the hashes of the samples, labeled with the corresponding category by Packer Detector (described in Section 3.2.1) in the hope that other researchers will use it as a basis for further studies.

[11]http://www.s3.eurecom.fr/datasets/low_entropy_malware/LEM_dataset.7z

Table 3.8: Overview of Previous Approaches

Paper	**Type**	**Features**	**Dataset construction**
[LH07]	Ent.	Ent.	*Not Packed*: benign executables *Packed*: Packers manually applied to benign files
[CkKOcR08]	ML	PE Heuristics	*Not Packed*: benign executables and PE files from AV manually analysed *Packed*: benign executables and PE files from AV vendors manually analysed
[PLL08b]	ML	PE Heuristics Ent.	*Not Packed*: benign executables *Packed*: malware from MALFEASE project and application of a set of packers benign executables
[PLL08a]	ML	[PLL08b] N-grams	*Not Packed*: malware from MALFEASE project filtered with unpackers ([RHD+06], [KPY07]) *Packed*: malware from MALFEASE and benign files
[SUPS+11]	ML	PE Heuristics Ent.	*Not packed*: benign executables and malware from VxHeavens *Packed*: Variants of the 'Zeus' family and application of a set of packers to the benign executables
[UPSGF+14]	ML	[SUPS+11] N-grams	*Not packed*: mal/good-ware filtered by PEid, entropy analysis, IAT entries, imported dlls and ratio of standard sections *Packed*: Application of a set of packers to the benign files, malware reported by PEid as not packed, 'Zeus' family
[DN12]	ML	PE Ent.	*Not packed*: benign executables *Packed*: Application of UPX to benign executables
[UPSS+12]	Ent.	Ent.	*Not packed*: benign executables and malicious samples taken from VxHeavens and checked with PEid *Packed*: application of some packers to benign files ,malware from Zeus family
[STMF09]	ML	PE	*Not packed*: Benign executables and malware from VxHeavens/Malfease filtered by PEid *Packed*: malware detected packed by PEid
[RV15]	Ent.	Ent. Statistical	*Not packed*: benign executables *Packed*: packers/encoders on benign files
[LRK+19]	ML	Ent. Statistical	*Not packed*: benign executables and evaluation of similarity for adding other binaries *Packed*: Samples tested with [UPBSB15]
[HL09]	Ent.	Ent.	*Not packed*: benign executables *Packed*: malicious samples from honeypot
[TZ09]	ML	PE	*Not packed*: Windows files and files filtered by PEid *Packed*: malicious samples filtered by PEid
[ASPE13]	ML	PE Heuristics	*Not packed*: benign and malicious file from honeypots *Packed*: Malicious samples tested with [Ste05]
[JCN+12]	ML	Opcodes	*Not packed*: Benign executables fetched from Windows Installation and unpacked malware *Packed*: Executables packed with off-the-shelf packers

Chapter 4

The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study

While in the first chapter we demonstrated an example of how the human can help the binary analysis pipeline by redirecting it at the initial step, now we want to show a use case of human-in-the-loop approach where the analyst is in the middle between two algorithmic machines, and serves as a bridge to simplify the communication between the two parts. We do this by concentrating on a different binary analysis task, i.e., vulnerability discovery, and by presenting a novel methodology to detect vulnerable flaws in binary executables.

As our world continues to accelerate into a software-powered future, vulnerabilities in the software that supports our lives are on the rise. This poses a set of unique challenges for software development and testing. Software tends to be checked for bugs by two categories of testers: by those developing it and thus having access to the source code(source-level program analysis) and by external security researchers who, often, do not have access to the source code (binary-level program analysis).

Source-level vulnerability analysis is fundamentally different from binary-level vulnerability analysis, because critical information about the software, such as type, structure, and size information, is lost when the software is compiled. This makes performing certain analysis paradigms, such as static vulnerability detection, on binary code a daunting challenge: before vulnerabilities can be detected in binary code, this lost information must be somehow recovered. This explains why little work exists in this direction [cweay] and why commercial tools that can analyze binary code (such as Veracode) require the application to be compiled with debugging symbols [veray] (i.e., inherently requiring the source code). Lack of source code also hampers other analysis paradigms, such as fuzzing and symbolic execution, because even these techniques benefit from the ability to *compile*, rather than retrofit, instrumentation into the analysis target [PF20]. As a result, static analysis techniques tend to require source code to effectively detect vulnerabilities, and dynamic techniques also function better when source code is available.

Interestingly, there is a related area of research that concerns itself with recovering information lost in the compilation process: *decompilation*. In recent years, techniques have been proposed to improve the recovery of data types [LAB11, NLC16], code structure [YEGPS15, YDGPS16, GDFFA20], and even exact syntactic identity [SRN+18]. These techniques have been integrated into increasingly powerful, accurate, and publicly available decompiler prototypes [KRS18, KOGY19, FCL+19].

Our insight is that the place, conceptually, where decompilation leaves off is *close* to the place where vulnerability detection picks up. That is, we realized that the type information, structure information, and pseudocode recovered by decompilers could be analyzed by vulnerability detection tools in lieu of the original source code, to at least some degree of efficacy. Additionally, as emerging techniques continue to improve decompilation results, and the gap between the original code and pseudocode from the decompilation of the program binary narrows, decompilers can become a more and more effective "crutch" to source-based vulnerability detection techniques.

Research questions

In this thesis, we undertake a study to determine the ability of current *Static Application Security Testing* (SAST) tools to detect vulnerabilities when executed on decompilers' generated code. While it might seem obvious that decompiler code is still unsuitable for static analysis, our case study wants to quantify experimentally how "far" we are from the point in which static analysis tools could be an effective solution on decompiled code. To do this, we measure precision and recall of 8 state-of-the-art SAST tools as they operate on the original code of 9 real-world applications versus the pseudocode of those applications resulting from the decompilation by 3 different state-of-the-art decompilers.

In summary, this chapter tries to investigate the following research questions:

- Can we "connect the dots" between decompilation and source-level static analysis?
- What is the effort required to the human analyst who wants to perform this vulnerability discovery task?
- How does the detection efficacy of the SAST tools change when we execute them over pseudocode?
- What are the root causes of these efficacy changes?

Our study has resulted in four main findings. First, the *output of current decompilers is unsuitable for any analysis by most SAST tools* without human analyst intervention and must be fixed before compilation-based analyzers (e.g. such as those based on LLVM passes) can be applied. In other words, in our pipeline, an analyst corresponds to the middle point between the decompilers and the SAST tools and her goal is to enrich the output of the decompilers to properly feed the static analysers. Second, when the compilation issues are fixed, SAST tools operate at a reduced 71% rate of recall suggesting that a latent potential could actually exist in this approach. Unfortunately, the precision of SAST tools on pseudocode suffered, with an average false positives increase of 232%. Third, compiler optimizations (especially function inlining) can sometimes *help* (and, at other times, hamper) SAST tools. Fourth, by analyzing discrepancies in SAST results between original and decompiled code, we detailed 7 root causes that impact the differences between false positive and true positive detection performance.

In turn, a number of immediate steps forward can be inspired by our results. Our research solidifies an idea that *modern decompilers are designed to generate pseudocode that is easy to understand for humans*, while SAST tools are not designed to ingest it. This suggests a set of new directions for researchers: small improvements to decompilers can improve the efficacy of SAST tools on *binary* code, even though they were designed with a source code requirement in mind. Alternatively, future studies could focus on SAST tools to make them more noise-resilient when parsing decompiled code. For example, the fuzzy-parsing approach performed by Joern [YGAR], already goes in this direction. Furthermore, the use of decompilers as a first stage in source-level static analysis can have applications beyond the use of SAST tools on our dataset. For example, embedded device firmware remains difficult to test with either dynamic (due to the *rehosting problem*) and static (due to the binary-only form in which the firmware is often distributed) techniques. Though some limited progress on both fronts has been made [KKK+20, DPY18, PGG+15], decompiler-aided static analyzers could automate vulnerability assessment for these scenarios where no standard alternatives exist.

4.1 Related Work

4.1.1 SAST

Static Application Security Testing (SAST) aims at removing vulnerabilities from the source code during the development phase as argued by Chess et al. [CM].

The first approaches consisted of a simple lexical analysis of source code to detect the presence of known vulnerable constructs [ratay, VBKM, PSDV06] (e.g., dangerous API invocations).

To overcome the limitations of these naive techniques, researchers proposed new approaches that leveraged a more detailed model of the source code, often obtained by relying on the compilers parsing components such as the AST generation [YLR12, LLZW17, SEHM13].

Other researchers instead tried to improve the detection accuracy for specific classes of bugs. This was the case, among the others, for buffer overflows [WFBA00, GJC+03, HDWY06, XGM08, KLHC10], use after free [YZH14, FMB+19, YSCX17, YSCX18], and null pointer dereference [HSP05, HP07, MJZ+15].

That being said, the case study that we present in the current chapter is closer to the many studies that focus on benchmarking program analysis tools, such as [EN08, CK11, Kra05, McL12, AM14, ACC+17, FBH18], where several aspects are analyzed ranging from the creation of a comprehensive testcase to a different set of tools adopted in the experiments.

4.1.2 Decompilers

One of the first studies about decompilers was conducted in 1995 by Cifuentes [CG], as part of her Ph.D. dissertation where she described how a decompiler works, the future challenges in the field and presented *dcc*, a decompiler for Intel 80286.

Over the past two decades, two main approaches have emerged for the development of decompilers: rule-based decompilation and NMT-based (Neural Machine Translation) decompilation. Rule-based approaches [ghiay, hexay, BLSW13, KMZ17] are the most popular today even though the production of a rule-based decompiler is particularly time consuming. For instance, according to its authors, the development of RetDec took a total of 7 years for a team of 24 developers [avaay].

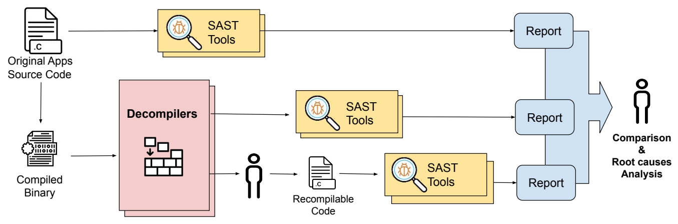


Figure 4.1: Summary of the Experiment Pipelines

The birth of the NMT-based approaches [KRS18, KOGY19, FCL+19] coincides with the seminal work of Katz et al. [KRS18], where the authors generalize the decompilation problem as a language translation task, namely from assembly to C thanks to the adoption of Natural Language Processing (NLP).

Another line of research has focused on improving the quality of the generated decompiled code by focusing on two main aspects: improving the readability and improving the control flow layout. The first category includes work that aimed at better recovering variable types [LAB11, NLC16] and at suggesting more meaningful variable names [LYS+19]. The second category traditionally focused on reducing the number of GOTO statements generated by the decompiler [YEGPS15, YDGPS16, GDFFA20] (*DREAM/DREAM++* decompilers) .

It is important to underline that all these studies focused only on improving the readability (and therefore the usability of the decompiler output) for humans. No study to date has analyzed how easy it is for a machine to process the produced code.

Finally, in 2018 Schulte et al. [SRN+18] proposed a novel approach to generate a binary-equivalent decompiled code that can be successfully recompiled. The paper by Schulte relies on a number of innovative techniques, such as adoption of existing decompilers to seed the lifting process and use of a human-written code excerpts for the generation of human-readable code even though the tool (named BED) is not released.

4.2 Experiment Design

This chapter studies how modern static analysis tools are impacted by the decompilation process, from the perspective of vulnerability detection. To that end, we study the interaction of the following entities: SAST tools (Sec. 4.2.2), Vulnerable applications (Sec. 4.2.1) and Decompilers

(Sec. 4.2.3).
For each vulnerable application, we proceed as summarized in Figure 4.1 , where two main pipelines are executed.

Baseline analysis. In the *source code analysis* pipeline, we input the original source code of the application to the different static analyzers and store their generated reports for later analysis.

Compilation. We compile each application according to the provided build scripts (e.g., Makefiles), using the same compiler options as suggested by the developers, to obtain the *compiled binary* that is in turn fed into the *decompiled code analysis* pipeline. A further insight is presented in Section 4.3.8, where we show the results of the differential analysis we performed for a subset of the vulnerable applications to assess the impact of compiler optimizations.

Decompilation and analysis. In the decompiled code analysis pipeline, we decompile the binary using our decompilers and run the resulting code through the SAST tools that do not require re-compilation.

As we will describe in more details in Section 4.2.2, the majority of the SAST tools require to compile the target application (for example, to perform LLVM passes). Therefore, since the decompilers typically generate C-like pseudocode which cannot be re-compiled out of the box, we manually applied the fixes needed to make the decompiler result compilable by both the `gcc` and `clang` compilers. This time-consuming process is interesting for different reasons. First, it allowed us to complete the experiments with all the static analysis tools selected in our study. Moreover, it provided us with an invaluable feedback on the steps an analyst should take if they want to apply source-code static analysis on binary programs. In other words, it allowed us to quantify the feasibility and effort required by a *human-in-the-loop* solution.

After manually repairing the decompiled results, we process the *recompilable* code by the compilation-based SAST tools.

Result comparison. Finally, we proceed to manually compare the three sets of reports obtained in our experiments (the one on the original source code, and the two on the decompiled and recompilable code) to assess how the detection and false positive rates were affected by the previous steps. The results of this comparison are presented in Section 4.3.

Whenever results differ (i.e., if a previously detected vulnerability was no longer detected or if new false alarms were generated by the tools), we performed a *root-cause analysis* to determine the cause. This step, again performed manually, required us to progressively modify the decompiled code by making it more and more similar to the original source, until the effect we wanted to study disappeared (i.e., the vulnerability was detected or the false alerts were not raised anymore). We discuss the findings of this analysis in Section 4.4.

In the rest of this section we discuss the methodology we used to select vulnerable applications, SAST tools, and decompilers. It is important to note that the applications and SAST tools had to be selected together. In fact, to have enough results for our comparison, we required each vulnerability to be detected by at least two SAST tools, and viceversa. This constraint forced us to perform a long pre-selection phase in which we evaluated many candidates (both for vulnerabilities and static tools).

4.2.1 Vulnerability and Application Selection

Our selection of vulnerable code was driven by five main requirements.

Codebase size. We included a mix of small and large code bases to assess the impact of code complexity w.r.t. decompilation and vulnerability detection.

C++. We included a C++ codebase to evaluate the fact that decompilers only produce C code as output.

Real vulnerabilities. We collected real-world CVEs/bugs that are representative of the typical classes of bug. This would allow us to be as general as possible in the evaluation phase, without focusing on artificially generated vulnerabilities.

Bug complexity. Third, an important factor that affects the precision of static analysis is whether the bug that needs to be detected is *inter-procedural* (i.e., its discovery involves to go through multiple functions) or *intra-procedural* (i.e., it is self-contained in a single procedure) We included examples of both categories, with a preference for *intra-procedural.* In fact, the purpose of our testbed is not only to benchmark SAST tools, but to cover bugs with different detection complexity.

Bug discoverability. Finally, we were also limited by the fact that our vulnerabilities should be identified on the original code by the SAST tools, to compare with the decompiled code output.

To satisfy our constraints, we collected 10 vulnerabilities from nine applications (summarized in Table 4.1). The applications ranged from 4 Thousands to 2.1 Millions LOC (for LOC statistics see Tab. 4.4). Note that for two

Table 4.1: The vulnerabilities adopted for the evaluation

Vulnerability	Application	Description
CVE-2017-1000249	file (C)	Stack BOF, unchecked memcpy
CVE-2013-6462	Xorg component (C)	Stack BOF, unchecked scanf
BUG-2012	libssh2 (C)	IOF (leading to heap BOF)
CVE-2017-6298	ytnef (C)	NPD
CVE-2018-11360	wireshark (C/C++)	Heap BOF (off-by-one) (*)
CVE-2017-17760	OpenCV component (C++)	BOF in C++ virtual method
CVE-2019-19334	libyang (C)	Stack BOF, unchecked strcpy
CVE-2019-1010315	wavpack (C)	DBZ
BUG-2010	libslirp (C)	UAF
BUG-2018	wireshark (C/C++)	DF (*)

(*) indicates an inter-procedural bug, all the others are intra-procedural projects,`Xorg` and `OpenCV`, the vulnerability was present in a sub component of the application that could be compiled as an independent module. Our dataset covers the following five classes of vulnerabilities:

Buffer Overflow (BOF) are probably the most widespread class of vulnerabilities and this is why we decided to include five variations of it, e.g., three incorrect uses of a buffer handling API (respectively `scanf`, `memcpy` and `strcpy`), an example of heap-based off-by-one buffer overrun (interprocedural) and finally a further stack-based BOF, present in a C++ code base and located in the implementation of an abstract method from a parent class.

Integer Overflow (IOF) bugs are a common cause of undefined behaviors in software. Our dataset includes one example of IOF that affects the size of a dynamic memory allocation, and that therefore can lead to an heap BOF.

Null Pointer Dereference (NPD) bugs exist when a NULL pointer is dereferenced. We include one example of NPD in our data set: in this example the pointer is returned by a `calloc` invocation and it is stored inside the field of a structure. The bug is due to the fact that the caller fails to check the pointer validity.

Double Free/Use After Free (DF/UAF). On the one hand, we would expect that such vulnerabilities are easier from the decompilation point of view, because the decompiler can reconstruct the use of a `free` without any type system/size problems. On the other hand, SAST tools that detect DF/UAF need to internally keep track of freed pointers and check the subsequent pointer accesses. As a further layer of complexity, one of the two bugs (the DF), is the second of the two inter-procedural vulnerabilities.

Division By Zero (DBZ) is not a memory corruption bug, but it affected several real world software in the past and can be used as denial of service vulnerability.

4.2.2 SAST Tools Selection

Firstly, we wanted to evaluate a range of products relying on a diverse set of features and techniques. Therefore we identified twelve tools (nine open source and three commercials) based on their popularity according to the studies proposed by [Kra05, SMM15, CK11, MM18, FBH18] and including non-academic sources such as [sasay, bloayb, bloaya].

Over the 12 candidate SAST tools, we selected those ones that were able to satisfy the selection criteria of detecting at least two of the vulnerabilities

Table 4.2: SAST tools selected for our study

	Commercial / Open Source	**Compilation required**	**User-provided queries**
CPPCheck	Open Source	No	No
Joern	Open Source	No	Yes
Fortify	Commercial	Yes	No
Infer	Open Source	Yes	No
Clang	Open Source	Yes	No
Ikos	Open Source	Yes	No
Code-ql	Open Source	Yes	Yes
Checkmarx	Commercial	No	No

in our dataset (cf. Table 4.3 detailed in Section 4.3).[1] Finally, our collection of static analyzers, listed in Table 4.2, includes: **CPPCheck** [cppay] 2.1, **Joern** [joeaya, YGAR] 1.1.95, **Fortify** [foray] 19.1.2, **Infer** [infay] 0.17.0 , **Scan-build** [scaay] 11.0, **Ikos** [ikoay] 3.0, **Code-ql** [codaya] 2.2.4, and **Checkmarx** [cheay] 9.2.[2].

Before selecting these eight tools we conducted a set of preliminary experiments, in which we tested many other SAST tools such as Coverity [covay], Frama-C [fraay], CPAChecker [BK] and Flawfinder [flaay]. However, we discarded them because after executing on a subset of bugs, they did not show a sufficient detection rate.

4.2.3 Decompiler Selection

We selected three cutting-edge decompilers for our evaluation: HexRays 7.1 [hexay] (the state of the art commercial decompiler from IDA Pro), Ghidra 9.2 [ghiay] (the leader open source decompiler), and Retdec 4.0 [KMZ17] (the emerging challenger).

Two main reasons influenced our choice of these three tools. First, other emerging alternatives are quite far behind in terms of precision and quality of the generated code. Furthermore, prior work about decompilers [YEGPS15, YDGPS16, GDFFA20, SRN$^+$18] only focused on these three decompilers when performing their evaluations.

[1] Note that the low detection rate of some tools may just be due to their underlying strategy in minimizing the false positive rate.

[2] The name of the commercial tools is provided for reviewing purposes and it will be anonymized before publication

Table 4.3: A breakdown of the bug detections for SAST tools (when applied to original program source code).

	CPPCheck	Joern	Checkmarx	Clang	Ikos	Infer	Code-ql	Fortify
CVE-2017-1000249	✓	✓	✓	✓	✗	✗	✓	✓
CVE-2013-6462	✓	✓	✓	✓	✗	✗	✓	✗
BUG-2012	✗	✓	✗	✓	✓	✓	✓	✓
CVE-2017-6298	✗	✓	✓	✗	✓	✓	✓	✗
CVE-2018-11360	✗	✗	✗	✗	✓	✗	✓	✗
CVE-2017-17760	✗	✓	✗	✓	✗	✗	✗	✗
CVE-2019-19334	✓	✓	✗	✓	✓	✗	✓	✓
CVE-2019-1010315	✓	✓	✗	✓	✗	✗	✓	✗
BUG-2010	✗	✓	✗	✓	✗	✗	✓	✓
BUG-2018	✗	✗	✗	✓	✗	✗	✓	✗

Non-decompiling lifters. Some tools, such as MCSema [DR14], can lift binary code directly to LLVM IR, in lieu of decompilation. At first glance, these might be a usable route for applying compilation-requiring SAST tools on binary code. However, these tools perform only a subset of the analysis which are executed by decompilers, and, in fact, can be considered as the "first stage" of a decompilation process. As a result, their output will contain insufficient information compared to the result of a decompiler, making the resulting code unsuitable for SAST analysis. For example, bytecode produced by lifters does not contain debug information whereas SAST tools that work on top of an llvm pass typically need compiler-generated symbols. Though it would be possible to develop more sophisticated SAST tools that bridge the gap between the output of static lifters and their expected input, this is exactly what decompilers already do from the other direction.

4.3 Experiments

In this section we discuss our experiments with particular focus on how the decompilation process affects the detection and false positive rates. We leave the investigation of the root causes to Section 4.4.

4.3.1 Source code analysis

Table 4.3 reports the detection results of the eight SAST tools when analyzing the original source code of the vulnerable applications.

It is interesting to notice that, except for Joern, Clang and Code-ql, the other tools are quite complementary in the bug detection, uncovering 2-4 bugs each and missing only two bugs overall (CVE-2017-17760 and BUG-2018).

Table 4.4: LOCs produced by different decompilers

Application	Original	HexRays	Ghidra	RetDec
file	14,056	18,012	18,296	24,114
Xorg	20,331	32,131	31,132	62,819
libssh2	22,322	26,806	33,186	37,531
ytnef	4,025	3,529	5,736	4,427
wireshark	2,110,822	2,345,564	2,444,145	*NA*
openCV	507,508	826,104	871,848	*NA*
libyang	102,750	104,789	98,886	151,049
wavpack	6,084	8,671	11,645	23,084
libslirp	7,806	12,178	14,058	15,915

The high detection rate of Joern and Code-ql is due to the custom query rules written by us and inspired from the guidelines described by the authors [joeayb, codayb]. Although our scope was not to generate a query that is sufficiently generic to cover the many possible scenarios for a certain class of vulnerabilities, we tried to put ourselves in the position of an analyst who does not know the bug a priori and this explains why the user-defined rules still generate a number of false positives. [3].

Even though our effort was to produce generic rules, it is unavoidable to introduce some bias. However, note that this is the only way to include the two analyzers, that represent the current state-of-the-art w.r.t. SAST. Making the queries more generic to catch a broader set of vulnerabilities for a specific class of bugs would also result in a biased result, by increasing the false positives. The opposite strategy (i.e., extremely dedicated queries that only capture the bug under testing) would not be representative of rules that can be used in the real world.

The remaining six analyzers were launched with their own set of rules and thus they do not introduce any bias in the experiment. In particular, we decided not to create custom rules for other tools (such as Checkmarx or Fortify) as they are already shipped with a full set of rules that were sufficient to detect some of the vulnerabilities in our dataset.

4.3.2 Decompilation

All three decompilers were able to successfully decompile the nine binaries in our dataset, except for RetDec which failed on the largest projects (Wireshark and OpenCV) due to LLVM errors. To measure the accuracy of the generated pseudocode, we draw inspiration from the authors

[3]queries are presented in our anonymized repository at `https://anonymous.4open.science/r/dae156f4-5332-4a06-a27e-1e7fac2b4d23/`

of [YEGPS15, GDFFA20], who adopted LOCs and number of GOTO statements to compare the different decompilers outcome in their work. As a coarse-grained indicator, Table 4.4 reports a comparison of the lines of code. The output of HexRays was the smallest in most experiments, and in total resulted in 20.8% more LOCs with respect to the original source files. Ghidra's code was not too far (+26.2% over the original), while RetDec was considerably more verbose (+79.8% in the binaries in which it ran successfully).

Previous papers often counted the number of GOTOs to measure the 'quality' of the produced code. While quality was often used as a synonym for readability, and it is unclear whether this would have any affect on SASTs, a lower number of GOTOs could also be considered a sign of a more advanced decompiler. We noticed that all tools generated code containing many GOTOs, ranging from a minimum of 84 (HexRays on `ytnef`) to a maximum of 36,002 (HexRays on `Wireshark`). In average, HexRays generated one GOTO every 60.3 LOCs (of the original source), Ghidra one every 60.7, and RetDec one every 11.2 LOCs.

Finally, we compared the function declarations of the projects source code against the ones contained in the pseudocodes produced by the three decompilers in order to measure the difference in the number of input parameters. On average, HexRays misses 4 parameters, Ghidra 6, and RetDec 7 every 10 function declarations.

4.3.3 Human role

Three among our SAST tools can directly analyze source code files without any need to compile them: CPPCheck, Joern and Checkmarx. The first two were able to analyze the output of the decompilers, without any further manual intervention. Checkmarx instead failed at reconstructing the AST for five instances of decompiled code.

Furthermore, the remaining five tools require the compilation of the target application to analyse it. However, as shown by the authors of [LW20], *none* of the output produced by the three decompilers was correct C code, and therefore none of them could be re-compiled out-of-the-box. This obliged us to look for a suitable solution to continue our experiments.

Therefore, to put ourselves in the position of an analyst, we attempted to manually fix the produced pseudocode to make it compliant with both GCC and Clang. We performed this operation on the output of all the three decompilers considered in our study, to compare different executions of the static analyzers on different input pseudocodes.

Overall the manual procedure took from a minimum of 90 minutes to 8 hours (for `libyang`). However, after spending 24 hours each by trying to fix the decompiled code of Wireshark and OpenCV (the two largest projects), we could not obtain a "recompilable" version of the pseudocode. Hence, for these two applications we adopted an alternative solution, that allowed us to generate a version of the decompiled applications that preserved the vulnerabilities and could be processed by our SAST tools. In particular, for these two cases we fixed the pseudocode of the vulnerable functions and of all of the procedures they invoked. We then integrated the resulting code into the the original source code of the vulnerable module – thus resulting into an hybrid codebase where all code related to the vulnerability came from the decompiler while the rest was taken verbatim from the original codebase of the module. This compromise allows us to study whether SAST tools could still find the vulnerability in the recompilable code, extending our evaluation of the tools to all the pre-selected vulnerabilities, but not to measure the impact on the overall number of false positives.

Our manual procedure consisted of a number of repeated steps that involved the proper definition of global variables, the definition of header files, the correction of function invocations (e.g., often the decompiler declared a method with N parameters and invoked it with $M \; ! = N$ parameters), the resolution of mismatching types, and some small syntactic operations to remove wrong keywords or fix syntax errors with brackets.

Although we are aware of the fact that some bias could be introduced while manually fixing the pseudocode, we want to underline that this mimics a realistic setting since currently a human-in-the-loop solution is required for this approach and alternatives are still missing.

4.3.4 Decompilers variability

The detection outcomes of the SAST tools able to analyze the output of the three decompilers is presented in the 'Decompilers Output' columns of Table 4.6. These outcomes are not broken down for each of the three decompilers as, except for the case of CVE-2017-6298 discussed below, the detection results were always the same regardless of the decompiler.

Indeed we launched the 8 static analyzers for each version of the decompiled code (either the raw or the manually fixed one depending on the tool). Unfortunately, some combinations of analyzer-pseudocode could not produce an analysis result because the corresponding tool failed with a crash. Except for an execution of Ikos on the Hex-Rays decompilation of CVE-2019-1010315, the other exceptions affected mostly the output of Ghidra and Retdec when analyzed by Ikos (3 failures on Retdec, 5 on Ghidra), Fortify (2 failures on Retdec) and Checkmarx (3 failures on Retdec, 2 on Ghidra). For all the other tools instead, it was possible to compare the output in terms of detections, finding that no differencies exist between the HexRays and Ghidra outcome from the SAST perspective.

The same does not hold for the output of RetDec. Overall the code generated by RetDec was more complex and considerably less readable for a human analyst. However, readability does not necessarily affect automated algorithms, and in fact vulnerability CVE-2017-6298 could only be detected on the RetDec output when using Joern and Code-ql. This is due to the fact that RetDec adopts a more naive approach and represented the fields of a struct as if they were separate variables (while both Ghidra and HexRays reconstructed a struct), before assigning them in the pseudocode representation of the struct (i.e., an array). As we will explain in more details in Section 4.4, this helps static analysis tools to more easily track the use of the individual fields, which in the aforementioned case helped to discover the vulnerability.

We searched for other cases containing structs to see if they also benefited from the RetDec decompilation approach, but neither the Use-after-free nor the Double free bugs that are related to struct usage could be discovered on the RetDec decompiled code. Note that since RetDec failed to decompile Wireshark in its entirety, we manually tried to point the tools directly to the vulnerable functions (which were decompiled by RetDec), but this did not lead to any detection because in those cases the generated code was more similar to the HexRays one and it contains some patterns that make the bug detection harder. As we will explain more in details in 4.4, the representation of types and structs in the pseudocode is crucial for SAST tools.

In the rest of the thesis we consider a bug as detected by a static analyzer on a binary if at least one decompiled code exists such that the tool can identify the vulnerable flaw when analysing it. Similarly, due to space limitations, for Table 4.6 (where we evaluate the variation of false positives), we only report results on the HexRays decompiled code. Moreover, given the failure conditions that some tools experienced on Retdec and Ghidra, the false positives evaluation on these would be incomplete.

4.3.5 Summary of Results: True Positives

Table 4.5 presents a summary of the results, both for the tools that we were able to run on the vanilla output of the decompilers, as well as for the remaining ones that we had to test on the manually curated code. The green marks represent the cases where the bug was found on the pseudocode, whereas

68

Table 4.5: Results of running the SAST tools over decompiled code. An asterisk (*) signifies that the detection was accomplished through the introduction of an excessive amount of false positives.

	Decompilers Output		Re-Compilable Code						Wrt source
	CPPCheck	Joern	Checkmarx	Clang	Ikos	Infer	Code-ql	Fortify	code
CVE-2017-1000249	✗	✓	✓*	✓*	✗	✗	✓	✗	4/6
CVE-2013-6462	✓*	✓	✓*	✓*	✗	✗	✓*	✗	5/5
BUG-2012	✗	✗	✗	✗	✓*	✗	✗	✗	1/6
CVE-2017-6298	✗	✓	✓*	✗	✗	✓	✓	✗	4/5
CVE-2018-11360	✗	✗	✗	✗	✓	✗	✗	✗	1/2
CVE-2017-17760	✗	✓	✗	✓	✗	✗	✗	✗	2/2
CVE-2019-19334	✓*	✓	✗	✓*	✓*	✗	✓	✓*	6/6
CVE-2019-1010315	✓	✓	✗	✓	✗	✗	✓	✗	4/4
BUG-2010	✗	✗	✗	✗	✗	✗	✗	✓	1/4
BUG-2018	✗	✓	✗	✗	✗	✗	✗	✓	0/2 + 2
Wrt source code	3/4	6/8 + 1	3/3	5/8	3/4	1/2	5/9	2/4 + 1	

the cross marks show a missing detection. Dashes instead indicate that the bug was not found in both the original source code and the decompiled one.

We must underline that for five executions on the raw HexRays decompiled code, Checkmarx failed at building the AST of the analyzed code. For this reason, we opted to run it on the re-compilable code and to report the results related to such executions.

Overall, only one of the tools (Chechmarx) was able to re-discover the same subset of vulnerabilities as when it was applied to the original source code. However, all tools were still able to discover at least one bug (and often more than one), thus showing that running SAST tools on decompiled code is not a useless procedure. In total, the 42 cumulative True Positives on the original codebase decreased to 30 (71%) after decompilation. However, not all tools were equally affected, as reported in the last row of the Table.

The three tools that operate on source code without the need to compile it were less affected by the decompilation process. Moreover, the commercial tools, while in general less effective at discovering the vulnerabilities in our dataset, continued to find exactly the same bugs also in the decompiled code, even though in the case of Fortify, we can observe that a new vulnerability is uncovered instead of another that is not detected anymore. At the other end of the spectrum, Clang and Code-ql were the two tools that were affected the most by the decompilation process.

Another way to look at the data is to group the results in terms of vulnerabilities instead of looking at the different tools. In this case (all results reported in the last column of Table 4.5) the integer overflow (BUG-2012), the use-after-free (BUG-2010), and the double-free (BUG-2018) clearly stand out as the most difficult to detect on decompiled code.

Table 4.6: False positives, as evaluated on the original source code and the decompiled source code produced by Hex-Rays. False positive increases of over 50% are highlighted in red, and decreases are highlighted in green.

	Decompilers Output				Re-Compilable Code											
	CPPCheck		Joern		Checkmarx		Clang		Ikos		Infer		Code-ql		Fortify	
	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec
CVE-2017-1000249	68	45	166	144	236	241	88	96	3	9	152	423	174	515	202	583
CVE-2013-6462	106	503	374	296	485	701	368	365	303	359	89	812	496	1,177	246	1,097
BUG-2012	68	262	253	110	107	358	298	178	103	541	21	827	102	341	613	453
CVE-2017-6298	16	125	25	39	52	66	15	10	329	439	10	80	18	132	84	137
CVE-2018-11360	3,113	20,548	3570	1504	-	-	-	-	-	-	-	-	-	-	-	-
CVE-2017-17760	475	7668	122	1277	-	-	-	-	-	-	-	-	-	-	-	-
CVE-2019-19334	833	1072	1019	334	2,295	384	185	315	13	113	128	959	98	176	1999	661
CVE-2019-1010315	20	143	214	150	173	358	65	45	20	-	295	361	428	208	1417	806
BUG-2010	69	297	190	41	102	114	2	22	31	32	23	168	37	350	115	525
BUG-2018	3,113	20,548	3570	1504	-	-	-	-	-	-	-	-	-	-	-	-
Total	7,881	51,211	9,503	5,389	3460	2223	1,021	1,031	802	1439	718	3630	1353	2899	4676	4262

At the other end of the spectrum, the division by zero and the stack-based buffer overflows seemed instead the easiest to detect. For the first, a manual inspection shows that there are no interesting variations in the way the decompilers reconstructed the source code. The bug involved two integer variables which are easier to handle than strings/pointers for decompilers. Thus, after decompiling the corresponding binary, the pseudocode surrounding the vulnerability was quite similar to the original code, from a static analysis perspective.

For the three stack-based BOFs, the true positive instead came at the expense of a much larger number of false positives, as we will describe in more detail in the following section. For these cases we reported an asterisk (*) meaning that an high number of buffers' operations were flagged by the analyzer, partially explaining the detection for these cases.

4.3.6 Summary of Results: False Positives

The usability of a tool is largely determined by the number of false positives, since reporting thousands of alarms would make the triaging phase both difficult and time consuming.

We performed a study of the false positive increment for each project where we could compare the outcomes of the tools on the decompiled code. Thus, we decided to focus on the Hex-Rays output, since is the one that was easier to parse for the SAST tools reporting only one failure for CVE-2019-1010315 (as explained in Section 4.3.4, 3 tools failed on the Ghidra/Retdec output). Moreover, it was not possible to have such a comparison on the Wireshark and OpenCV projects, because we could not recompile the decompiled code.

We report the variation of false alarms in Table 4.6, marking in red the cases in which there was an increase of more than 50% of false alarms, and in green when the number decreased. Overall, if we exclude Joern (which is a special case we describe below) in 78% of the tests the number of false positives increased. Even worse, in 61% of the tests the false alerts increased by more than 50%.

We point out that we manually went through the generated alarms that the static analyzers produced, to assess if they represented actual false positives. The only assumption that we did to accelerate the procedure, was that if the use of an API call (e.g., `strcpy` or `memcpy`) was safe in the source code, it could not become vulnerable in the pseudocode. Moreover, many false positives could be discarded in batch since they were related to uninitialized variables.

However, in some cases (mainly for Clang and Fortify) the tools generated less false alarms on the decompiled code. To figure out the reasons behind this, we checked the reports for those tools that reported a negative variation. One of the main reasons for this behavior is that many false alarms in the source code are due to `free`-related vulnerabilities (UAF, DF, stack variables freed). However, when analysing the decompilers, the SAST tools could not apply the same dataflow and, moreover, the decompiler changed the type of the variable containing the freed memory area, making the job of the analyzers much more difficult. Furthermore, several warnings reported the presence of badly terminated strings in the source code (i.e., strings without the proper null-terminated byte). Because of type confusion problems, the same problem could not be detected in the decompiled code.

To evaluate the Code-ql false positive rate, we adopted the default queries that are shipped with the installation. This allowed us to obtain unbiased results, compared to what we would get if we used the custom rules that we wrote to find the vulnerabilities.

Finally, Joern deserves a separate discussion as the tool does not come with any predefined rule, and all tests were therefore performed by enabling our own heuristic checkers for each project scan. Although these are certainly not a complete and generic set, they allowed us to have a reasonable evaluation of the false positives also for this static analyzer. Moreover, such a tool performs a fuzzy parsing of the code. Even if this feature makes Joern the perfect candidate to analyze decompiled code, we pay for this fact in some cases where it could not correctly interpret some pieces of code and skipped them without providing a complete analysis. As a result, the internal representation lacks some parts that could not be correctly parsed, and thus were not reachable by our queries. This resulted in a decrease of the query output, as only a portion of the code could be properly analyzed.

4.3.7 Bugs detected *only* on pseudocode

Our initial assumption was that running a SAST tool on decompiled code can *at best* detect the same bugs it would detect when it is used to analyze the original source code of the application (and more likely considerably less than that). Albeit our experiments show that for the majority of the analyzed scenarios this hypothesis is correct, we found one interesting case (BUG-2018) in which tools (both Joern and Fortify) could detect a vulnerability on the decompiled code, but not on the original codebase.

Compilers can affect the control flow of a program in such a way that it is impossible to recover exactly the original version. For example, as we will see in the next sub-section, sometimes compilers remove dead code or simplify boolean conditions for optimization reasons.

The double free vulnerability present in Wireshark (BUG-2018) is an inter-procedural problem, which is therefore harder to detect for static analysis tools. In fact, as reported in Listing 4.1, the vulnerability involves three separate functions that eventually invoke the `g_free` two times.

In the original codebase Joern was only able to reconstruct a subset of the flows that lead to `free`, thus missing the vulnerability. Similarly, the internal analysis performed by Fortify was insufficient to uncover the vulnerable flow in the original source code.

However, after checking the decompiled code, we noticed that, because of the `static` keyword, the compiler inlined different functions into a single body (the decompiled version of `val_from_unparsed`). This transformed the inter-procedural bug into an intra-procedural one, largely simplifying the task of detecting the bug. In fact, it turned out that both Joern and Fortify succeedeed at revealing the bug on the pseudocodes they could analyse.

```c
static void
string_fvalue_free(fvalue_t *fv)
{
    g_free(fv->value.string);
}

static gboolean
val_from_string(fvalue_t *fv)
{
    string_fvalue_free(fv);
    return True;
}

gboolean
val_from_unparsed(fvalue_t *fv, ...)
{
        string_fvalue_free(fv);
        ...
        return val_from_string(fv, ...);
}
```

Listing 4.1: Double free source code

4.3.8 Compiler Impact

Compilers support different optimization levels that modify the output of the compilation phase at the assembly level. Therefore, we opted to analyze how these compiler options can affect the results of the decompilation, and in particular, if such changes in the pseudocode are meaningful for the SAST tools.

To verify this, we performed an additional experiment based on the following four phases. **(i)** Selection: we selected two among our open source projects, `file` and `libssh2` (CVE-2017-1000249 and BUG-2012). The choice of the two projects was driven by the average size of their codebase and the meaningful number of detections. **(ii)** Compile with optimization levels: we compiled the selected projects with three different optimization levels, `O0`, `O2`, `O4` (`O0` disables all optimization passes whereas `O4` indicates that the generated code is highly optimized to improve execution speed). It shall be noticed that all the experiments discussed so far were performed using the default compiler optimizations specified in each project makefile (always `O2` for our applications). **(iii)** Decompile: we decompiled the three versions of the same binary with HexRays. **(iv)** Analysis: we launched all SAST tools on each decompiled outcome. This also implies that we had to manually fix all variants of the decompiled code to generate the recompilable versions required by many of our tools.

The first aspect we wanted to investigate is how the compiler options affect the number of false positives. All static analyzers ran on all versions except for Ikos that reported some problems when parsing the code compiled with the `O4` option. Hence, we discarded it, for the computation of the false positives.

For `libssh2`, the tools cumulatively produced 850, 2,421 and 1,606 false positives for, respectively, `O0`, `O2` and `O4`. For `file` we obtained instead 3,085, 2,275 and 2,984 alarms, depending on the compiler optimization. Such results show that there is no clear trend and it is unclear whether a more aggressive optimization of the code would cause more or less false alarms. However, the different amount of false positives for each compilation option means that the compiler actually has an impact on the generated decompiled code, and therefore, on the way SAST tools parse it.

We then inspected all reports generated by the tools, to determine if vulnerability detection is also affected by the compiler optimizations. For BUG-2012, we could not find any difference between the executions of the static analysis tools over the different versions of decompiled code. The only configuration that brought to a detection consisted of executing Ikos on the `O0` and `O2` versions of the code. After a manual inspection of the three flavours of pseudocode, we understood that the compiler optimization level does not affect the vulnerable function in a significant way except for a different number of declared variables(29 for `O0` vs 99 for `O4`).

CVE-2017-1000249 instead tells a different story. Indeed, when scanning the three versions, the tools reported different results depending on the compiler optimization. More specifically, with `O0` and `O2`, 4 tools out of 8 could detect the bug anyway. Surprisingly, the detection dropped to zero with the `O4` flag. To understand the reasons behind such a drastic change, we went through the decompiled code one more time. A first difference is that with the `O4` flag, multiple functions are compiled inline and thus, the vulnerable function becomes a part of a bigger one, hindering the SAST tools to analyze the data flow. Moreover, such a modification, affects not only the local defined functions of the binary, but also some library functions. Among the others, the `memcpy` invocation, originally contained in the code and root cause of the buffer overflow is replaced with an inline implementation that is ignored by the tools. Finally, an unsafe check on the buffer size that always evaluates true (because of a programming error) is removed due to the optimization reasons, as described in more details in Section 4.4. Cumulatively, these three aspects make the life of SAST tools remarkably harder, leading to an increment of the false negative.

Although this experiment cannot uncover in a systematic way all possible scenarios where the compiler influences the resulting pseudocode, these observations indicate that the compiler influences the decompilation phase w.r.t. both the false positives and false negatives.

Table 4.7: Decompilation inadequacies that inhibited SAST tool operation on our dataset, separated into *patterns* and their effect on analysis results. Depending on the pattern, these inadequacies can be repaired through improvements in decompilation techniques, SAST approaches, or both.

Pattern	**Effect**	**Project**	**Affected Tool**	**Repairer**
(P1) Buffer size	FP ↑	all	all except Joern	Both
(P2) Integer types	FP ↑	all	Checkmarx, Cppcheck	SAST
(P3) Unitialized variable	FP ↑	all	Clang, Infer, Ikos,Code-ql	Decomp
(P4) Function pointers	TP ↓	`libssh2`	Joern,Code-ql	SAST
(P5) Pointer as int	TP ↓	`ytnef`, `wireshark`, `libslirp`	Joern,Clang Checkmarx, Ikos,Code-ql	Both
(P6) Int wrong size	FP ↑	all	Ikos,Code-ql, Infer	Both
(P7) Simplified expressions	TP ↓	`file`	Cppcheck Fortify	SAST

4.4 Root Cause Analysis

We investigate now the root causes that resulted in the performance degradation of the SAST tools when executed on the decompiled output. For this purpose we gradually change the pseudocode by making it more and more similar to the original codebase, until either the tool reported the missing vulnerability or until the extra false positive disappeared.

Our findings uncovered seven main root causes, four responsible for false positives and three for false negatives. For each of them we discuss the specific elements in the code (hereinafter *patterns*) introduced by the compilation and decompilation process that degraded the SAST performances. Our patterns are summarized in Table 4.7, along with the projects and tools affected by that specific pattern. The column **Repairer**, indicates which component of the toolchain (decompiler, SAST tool, or both) is in the best position to mitigate/address the problematical pattern. In fact, while on the one hand decompilers could try to infer more information from the binary, on the other hand SAST tools can be designed with this limitation in mind and be more permissive when dealing with the pseudocode.

Finally, we underline that our purpose is to illustrate such root causes that result in the performance degradation of the SAST tools, rather than proposing potential remediations to these issues that will require future research in both the decompilation and SAST fields to be addressed.

P1 - Inability to Recover the Size of Stack Buffers

Effect: increase false alarms Repairer: Both

A large number of extra warnings in the SAST output was reporting the presence of BOFs. As an example, we propose the following excerpt from the `file` application:

```c
#define PATH_MAX 4200
FILE* list = fopen(outfilename+1 "rb");
char listbuff[PATH_MAX * 2];
memset(listbuff, 0, sizeof (listbuff));
fread(listbuff, 1, sizeof(listbuff)-1, list);
```

Listing 4.2: Source code of a safe buffer access

Looking at this code, it is quite evident that the `memset` and `fread` invocations are safe in this context, thanks to the proper use of `sizeof` operator. The decompiled code looks instead quite different:

```c
FILE* v212;
char* s1;
v212 = fopen(dest + 1, "rb");
memset(s1, 0, 0x2000uLL);
fread(s1, 1uLL, 0x1FFFuLL, v212);
```

Listing 4.3: Pseudocode of a (ex safe) buffer access

The `sizeof` operator is resolved at compile-time, and therefore the decompiler only sees the actual numerical values. Intuitively, one would expect that this makes the SAST's job easier because now the tools do not need to compute themselves the size value. However, the array definition has been replaced with a scalar variable (`s1`) declared as a `char*`, without any information about its original size. As a result, when the SAST tools analyze the decompiled code, they flag the two calls as two potential buffer overflows, since the memory area pointed by the `char* s1` variable has unknown size.

In other examples, different ways to access the buffer (e.g., by index buf[i]), resulted in different warning such as NPDs, still because of the missing size information of a pointer variable.

Discussion: Although the issue is quite evident, a proper solution is not as simple and it inherently depends on the way compilers generate the assembly code. In fact, even with a sophisticated stack analysis, the decompiler cannot infer if a memory area belongs to the same buffer or it represents a group of distinct variables (in particular when an element of a buffer is accessed by using an hardcoded index). While some heuristics could be used to infer the original size, e.g., by looking at loop iterations or initialization routines, the risk is that by relying on this information the decompiler can hide the presence of vulnerabilities.

P2 - Signed and Unsigned Integers

Effect: increase false alarms *Repairer: SAST*

Many numerical statements were flagged by the SAST tools as potential IOFs. At a closer analysis, this was caused by two main errors in the decompiled code.

An example of this pattern are functions that return an integer value, where a negative value is associated with an error condition. For instance, this is a snippet of decompiled code from Xorg:

```
sub_9840(__int64 a1) {
  unsigned int v2;
  ...
  if(ERROR_CONDITION) { v2 = -1;}
  return v2; }
```

Listing 4.4: Negative return value in the pseudocode

The v2 variable is used to store the return value and it is assigned to -1 in case of an error condition. However, v2 is erroneously declared by the decompiler as unsigned int, and thus, assigning a negative value, leads the SAST checkers to think that an underflow can occur within that variable.

P3 - Integer Operations on Uninitialized Variables

Effect: increase false alarms *Repairer: Decompiler*

Mainly due to a more complex and interprocedural data flow in the decompiled code, we noticed that many SAST tools reported an addition (or subtraction) as potentially dangerous when they could not determine if one of the operands has been initialized.

As an example, we fetched the following lines of code from the libyang pseudocode:

```
sub_12B3E (..) {
```

```
2    v2 = 10; v4 = 1;
3    .. // a lot of code including GOTOs, etc
4    sub_129CF(.., &v2, .., v4);
5  }
6  sub_129CF(.., unsigned __int16 *a2, .., unsigned __int16 a4
     )
7  {
8    unsigned __int16 v9 = a4;
9    ..
10   *a2 -= v9;
11 }
```

Listing 4.5: Integer underflow due to an uninitialized variable

The subtraction at line 12 is flagged by Infer and Ikos, because such tools cannot find an initialization statement for the operands. However, after comparing with the source code, we noticed that both the two variables are initialized. The key difference is that in the source code those variables are initialized just before calling the function, while in the pseudocode they are initialized at the very beginning of the program so that very likely the SAST tools lose track of their propagation because of complex data flow.

Interestingly, so far decompilation researchers mainly studied variable types recovery [DC09, LAB11, NLC16] and names generation [LYS+19], but no prior work focused on the "position" of recovered variables in the control flow.

P4 - Function Pointers

Effect: decrease detection rate   *Repairer: SAST*

BUG-2012, which affects **libssh2**, is presented by Yamaguchi et al. [YGAR] as a use case of Joern, but while such a tool can detect it on the original codebase, it misses it in the decompilers output. The main vulnerability consists of an IOF resulting from a sum whose value is used as input for a dynamic memory allocation. As a consequence, the IOF can produce an undefined dynamic memory allocation resulting in wrong memory accesses. For clarity, we report the snippet of code in the following listing:

```
ssh2_packet_add(SESSION session,char* data,..){
...
uint32_t namelen =
  libssh2_ntohu32(data+9+sizeof("exit-signal"));
channelp->exit_signal =
  LIBSSH2_ALLOC(session, namelen + 1);
memcpy(channelp->exit_signal,
       data+13+sizeof("exit-signal"), namelen);
}
```

Listing 4.6: libssh2 vulnerable code snippet

The `LIBSSH2_ALLOC` macro allocates `namelen + 1` bytes and returns the requested memory in the `exit_signal` buffer, that is eventually accessed. If `data` is under the attacker control, it is possible to craft that variable so that the sum `namelen + 1` causes an IOF bug.

Listing 4.7 shows the resulting pseudocode:

```
vulnerable_function(int64 a1, int64 a2, ...) {
...
int name_len = non_vuln_function(a2 + 21);
*(_QWORD *)(v24 + 40) =
  (*(__int64 (__fastcall **)(_QWORD, __int64))
  (a1 + 8))((int)(name_len + 1), a1);

memcpy(*(void**)(v24 + 40),
  (const void*)(a2 + 25), name_len);
}
```

Listing 4.7: libssh2 vulnerable decompiled code

Now we can notice that the macro invocation has been replaced with its actual value that corresponds to a function pointer stored in the `struct session` at offset 8 (namely, the macro is defined as `session->alloc(...)`). The decompiler casts the function pointer accordingly to the function definition, resulting in a more complicated structure of the invocation. The pointer cast is the reason Joern and Code-ql fail at detecting the bug in the pseudocode. Joern is not able to properly parse it, and thus it entirely skips the call. In this case no query exist that can reach the vulnerable path.

Code-ql actually parses the code correctly, but because of its internally used representation, the query used to find the original vulnerability does not work anymore. It is possible to write a new, and much more generic, query that still capture the bug — but the more general rule would cause an increased number of false positives.

Discussion: The root cause of the problem is that the function pointer invocation contains many casting operations, therefore hindering the static parsing of the code. However, we could easily solve the problem by instantiating a variable that can store the function pointer address, and then invoking it in a separate line:

```
__int64 (*fcn_ptr)(_QWORD, __int64) = (a1 + 8);
*(_QWORD *)(v24 + 40) = (*fcn_ptr)((unsigned int)(n + 1),
   a1)
```

Listing 4.8: libssh2 vulnerable decompiled code after the fix

P5 - Pointers as Integers

Effect: decrease detection rate *Repairer: Both*

For this pattern, let us focus on the CVE-2017-6298, a null pointer deref resulting from an unchecked `calloc` return value.

Reading the following snippet of code the vulnerability looks quite evident, and in facts different tools can detect it (Joern, Checkmarx, Ikos, Infer, and Code-ql):

```
variableLength* vl;
...
vl->data = calloc(vl->size, sizeof(WORD));
temp_word=SwapWord((BYTE*)d,sizeof(WORD));
memcpy(vl->data, &temp_word, vl->size);
```

Listing 4.9: Null pointer dereference

The first thing is that the variable `vl` is a pointer to a custom struct whose definition is unknown for the decompiler. The `memcpy` invocation itself is safe because the code writes the correct size into the dynamically allocated buffer, but the `vl->data` value is not checked for nullness, potentially leading to a null pointer dereference if `calloc` returns a null value.

When the code is decompiled with HexRays and Ghidra, we obtain the following code:

```
signed int* v9;
size_t v19;
void* v20;
...
(_QWORD *)v9[0] = calloc(v9[2], 2uLL);
v18 = sub_19B0(v4, 2);
v19 = v9[2]; v20 = *(void **)v9; v76 = v18;
memcpy(v20, &v76, v19);
```

Listing 4.10: Null pointer dereference Hexrays pseudocode

We can immediately note that the struct is represented as a signed integer pointer (identifier `v9`). The `calloc` return value is written in `v9[0]`, after casting it to pointer.

Although the tools comprehend that the return value is written inside a local variable, they believe that the assignment happens in a variable of type `signed int`. Because of such a type confusion problem, from now on the static analyzers are not interested anymore in the return value, stop to track the data flow for that path and go on with the analysis of other potentially vulnerable paths. Overall, they miss the connection between the returned pointer and its dereferences that occur in the following code.

If we look instead at the code generated by RetDec:

```
int64_t * v103; int32_t v105;
..
int64_t * mem5 = calloc(v102, 2);
*v103 = (int64_t) mem5;
int64_t v104 = fun_19b0(v19, 2, v28, v1);
int64_t v108 = v104;
...
memcpy((int64_t *)* v103, &v108, v105);
```

Listing 4.11: Null pointer dereference RetDec pseudocode

What makes this output simpler to analyze for static analysis tools is the fact that the return value of the `calloc` API is directly stored into a proper pointer, without any further cast or array access. Thus, tools are able to track the data flow and can therefore recognize the use of the pointer within the `memcpy`.

In this example we discussed the case of a returned pointer assigned to an integer variable, but the same issue happened several times when decompilers declared parameters as integers instead of pointers in the functions prototypes (e.g., BUG-2010 and BUG-2018, that are respectively the UAF and the DF).

Discussion: SAST tools seem to have problems tracking pointers that become integer and later pointer again. Learning from RetDec, the solution is just to back-propagate the type information. In other words, if a variable is later casted to a pointer and de-referenced, then this information should be used to redefine the variable type as a pointer.

For instance, it is sufficient to declare an intermediate variable of type `int*` instead of `v9` in the HexRays's output and all tools that were missing the vulnerability were able to correctly perform their taint analysis until they reach the `memcpy` invocation.

P6 - Integers of Wrong Size

Effect: increase false positives *Repairer: Both*

Decompilers often declare variables of the wrong size (e.g., double-word instead of bytes) and then rely on cast operations to ensure the type system coherency of their output statements. This behavior caused many SAST tools to generate false alarms due to the potentially erroneous pointer casting.

As an example we can consider the code snippets in Listing 4.12 (original code) and Listing 4.13 (decompiled code).

```
 uint8_t out[SIZE]; uint8_t tmpout[SIZE];
 for(j = 0; j < sizeof(out); j++)
```

```
    out[j] ^= tmpout[j];
```

Listing 4.12: Suspicious cast source code

```
  __int64 v22;   __int64 v26;
 for ( j = 0LL; j <= 0x1F; ++j)
  *((_BYTE *)&v22+j) ^= *((_BYTE *)&v26+j);
```

Listing 4.13: Suspicious cast decompiled code

In the original code, the elements are of type `uint8_t` (i.e., one byte each). In the decompilers output the two variables becomes 64-bit integers, which are later casted to `_BYTE` to perform the xor operation. Furthermore, the retrieval of the j-th element, is done through pointers arithmetic with type `uint8_t`.

A similar pattern appears very often in our experiments, with different source pointer types and using different types to perform the cast. While this pattern is similar to the *Pointers as Integers* (in fact, again the decompiler used integer variables to store pointers) here is the wrong size and the cast operation that cause false alarms instead of the inability to follow the data flow as in the previous pattern.

It is also interesting to note how, because of the initial declaration of the variables representing the arrays (`v22` and `v26` are integer types and not integer pointers), the pattern is reported by some SAST tools as a dangerous cast, rather than a buffer overflow.

On the other hand, if in the previous code, the two variables were defined as `__int64`* we would still observe an alert warning for a potentially unsafe memory access, converging in the case described for **P1**.

Discussion:

This is again a case of type confusion, less severe than the pointer case (as it cannot lead to missing real vulnerabilities), but somehow harder to fix. In fact, back-propagating information to mark variables as pointers is not sufficient, and correctly sizing all integers requires a more complex analysis and inference techniques.

P7 - Simplified Expressions

Effect: decrease detection rate   *Repairer: SAST*

This last pattern is quite unusual, but we report it as it is the cause of some missed vulnerabilities in the decompiled code. For our discussion we use the CVE-2017-1000249, a stack BOF present in the *file* project. The original source code is depicted in the following snippet:

```
if (namesz == 4 && ... &&
    type == NT_GNU_BUILD_ID &&
    (descsz >= 4 || descsz <= 20)) {
      uint8_t desc[20];
      memcpy(desc, &nbuf[doff], descsz);   }
```

Listing 4.14: Source code of the BOF

The memcpy is unsafe because of the wrong check that is performed before its invocation. In fact, the OR operator is used instead of the AND to check if the size (descsz) is in the appropriate range. The boolean condition always evaluates to True, and this is detected by some tools (such as CPPCheck) and reported as potential bug— which in this case it is and leads to a buffer overflow.

However, compilers are also able to detect that the condition is always satisfied and they can simplify the code accordingly. This results in the following decompiled code:

```
char v58;
if (v49 == 4 && ... && v28 == 3) {
  memcpy(&v58, a4 + v45, v16);
  }
```

Listing 4.15: Decompiled code of the BOF

The desc buffer is another example of the inability of decompilers to reconstruct stack-based arrays. But the key element for this pattern is that the wrong test on the buffer size is not present anymore. Since the compiler is not generating its corresponding assembly code in the first place, the decompilers have no way to recover it.

Once the clue that was picked up by static analysis tools (the wrong check on the buffer size) is removed, some tools failed to detect the vulnerability altogether.

4.5 Discussion and Conclusions

We can distill the findings of our experiments around four main points, showing that even though few obstacles still remain, we believe that future work will be able to overcome these issues focusing on both the decompilation side and the static analysis part.

1. The main impediment to the use of SAST tools on pseudocode is that the decompiled code cannot be re-compiled out-of-the-box. The recent paper by Schulte et al. [SRN+18] make us feel optimistic that this problem will soon be solved. However, so far, human analysts need

to manually fix the decompiled code, a process that can take just few hours for small applications, but that becomes prohibitively complex for large codebases made of million of LOC.

2. Once the re-compilable issue is solved, existing SAST tools can discover (in our experiments) 71% of vulnerabilities they were finding in the original code. While there is still a margin for improvement, this result already go beyond our initial expectations. On the negative side, the number of false positives often increased considerably, making the output of many tools difficult and time-consuming to navigate. However, even even if the FP increase is on average 232%, in 29/61 cases the FPs either decreased or did not significantly increase, demonstrating how our approach is still promising in many scenarios.

3. Both the compiler and decompiler transformations contribute to the final result in a complex way. Our experiments show that there is no linear trend, and in some cases more aggressive optimizations even simplified the job of the SAST tools. For instance, in two cases static analyzers were even able to discover one vulnerability they could not find in the original source code.

4. Today, decompilers are still designed to generate code that is easy to understand for humans, and SAST tools are still designed to parse "well-written" code that is not generated by a machine. This human-centric view could, and should, change in the future. In Section 4.4 we listed 7 root causes that explain the differences we observed in the results. We believe that many of the entries in our list could be solved, or at least mitigated, by improvement in either the decompiler or the SAST analysis (or both).

Chapter 5

RE-Mind: a First Look Inside the Mind of a Reverse Engineer

For our last investigation, we move from the vulnerability discovery domain to the discipline of reverse engineering, a common binary analysis task that very often requires human experts intervention. As we will see in the follow up of the chapter, this activity deserves attention as we can imagine binary reverse engineering as a human-in-the-loop system where the analyst lives at the last node of the pipeline, in charge of interpreting the representations obtained by her set of tools, before providing the actual output of the analysis.

Researchers of different fields have studied, from a cognitive perspective, how humans perform several relevant activities with the goal of better understanding, improving, or automating field-related processes. For instance, in the area of computer science, many experiments have been conducted to study the mechanisms behind human's decisions in several tasks, ranging from program comprehension [Let87, RTKM12] to human-computer interaction [Ban, Kap96], and from problem solving [LKJ+16, CFBS15] to computer security [MBC17, VSR+18]. The crossover between the human mind and computer science has also resulted in the creation, and in the recent rapid evolution, of the field of artificial intelligence.

On the one hand, fully autonomous systems have already replaced humans in several security-related tasks including, among the others, host and network-based attack detection [OEV+, ZEKAS17, THLL09], malware classification [UAB19, MDC17, PHL+15] and phishing detection [MHK08, JST13, AA14]. On the other hand, other areas are still mostly human-driven.

For instance, binary *reverse engineering* (RE) is still performed entirely by highly skilled security experts. Machines play an essential role in the process in the form of tools to unpack, disassemble, emulate, and perform binary similarity. However, humans are still responsible for "understanding" the code. That is, experts are the last step of an analysis infrastrcture made of an arsenal of tools that lift and smooth the original machine code and are in charge of providing the final outcome of the analysis. This requires considerable expertise, together with a long and tedious manual effort. Unfortunately, the limited number of expert reverse engineers in the world is insufficient to cope with our society's security needs and the continuous growth in the amount of released software. The recent DARPA Cyber Grand Challenge (CGC) drove progress in computers' ability to reason about program binaries autonomously and discover vulnerabilities. However, these programs are still far from being able to compete against RE experts[1].

To overcome this problem, we believe it is fundamental to first understand *how humans approach and solve static RE tasks*. The comprehension of the most effective RE strategies used by expert humans can drive further research in the development of automated approaches, but it can also help design tailored training programs that can increase the number and the effectiveness of our experts.

Let us use a simple analogy to introduce the motivation for our work. When a professional chess player decides her next move, she has hundreds of millions of possible combinations to evaluate. However, previous research on the human brain of chess players has shown that this is not the way she reasons. Her brain can instead recognize patterns and naturally focus only on a handful of possible "good" moves. Now think about an expert reverse engineer. Similarly to a chess master, she also does not "evaluate" every single line of assembly code in a program, but she just skims through the code, focusing only on those critical parts to understand the code's logic. We believe that her primary skill is not to read faster every single basic block, but instead that she does not waste time reversing the ones that are not important for her task. In other words, she can see patterns where others can only see endless lines of code.

Sadly, today we do not know whether this hypothesis or any other hypothesis about how RE experts think is correct. At the 2020 Usenix Security conference, Votipka et al. [VRM+20] presented the first human study about

[1]The 2016 DEF CON CTF final put the best DARPA cyber-reasoning system (Mayhem [DARay]) against human teams. The supercomputer ended up in the last position [defay] (even on simplified challenges explicitly written to accommodate the limited architecture supported by the machine).

RE. This work inspired our research, where the main focus is restricted to *static RE* (from now on used alternatively with RE) from the perspective of assembly code comprehension.

Research questions

Our goal is to investigate a set of hypothesis by means of quantitative measurements and statistical tests conducted on fine-grained recordings of real RE tasks. To recruit a sufficient number of geographically-distributed participants, we designed an online platform that mimics the UI of traditional interactive disassemblers. We then used our platform to record the fine-grained behavior of 72 reversers while they solved two different reverse engineering exercises. In total, we collected 272 hours of binary reverse engineering activity, which we then analyzed to identify patterns and strategies that we can use to model the 'experience' of a reverser.

The research questions that drove our study, methodology and experiments are the following:

- What do experts do differently from novices?
- Do experts/novices share particular strategies to explore binary code?
- How are these strategies linked to the binary code elements (e.g., functions, basic blocks)?
- Is any particular strategy correlated with better RE performances?

The results of our experiments allowed us to confirm that experts indeed visit less basic blocks than beginners, and they are also able to dismiss on average 22% of the blocks they visit in under two seconds. While novices tend to re-visit the same parts of the code multiple times, experts gain more information during their first visit. We also identified several exploration strategies, both at the basic block and the function level, that seems positively correlated with experience. For example, beginners are more likely to explore a binary 'horizontally,' while more skilled reversers are more likely to proceed in a vertical way.

These are only a few examples of the many features we investigate in this last chapter to characterize the static reverse engineering process. We believe that this fascinating area of system security, where human experience is highly regarded but little understood, can help our community to better understand the mechanics behind the cognitive aspects of reverse engineering.

5.1 Related Work

To the best of our knowledge, only four studies have been conducted so far on human behaviors in the context of reverse engineering [SKBM06, Bry12, VRM$^+$20, CC19].

One of the first studies was conducted by Sutherland et al. [SKBM06] in 2006 to demonstrate that the education/technical knowledge and the ability to reverse engineer simple binary files are positively correlated. In 2012, Bryant [Bry12] performed a semi-structured interview with the addition of in-place observations during the RE sessions to investigate four experts approaching typical RE scenarios, such as breaking the protection scheme of a toy binary. The outcome is a precise observation of the skills, mental flows, and knowledge-based techniques that the subjects exhibit while reversing a binary. Interestingly this work tries to be the bridge between the source code comprehension community and the RE one, by studying how reverse engineers make use of assembly patterns.

In 2018, Claire et al. [CC19] proposed RevEngE, a framework to monitor reverse engineering from several points of view. The framework is based on an instrumented virtual machine that registers events such as spawning a new process, focusing on a window and mouse clicks. The goal of this work was to describe a system that acts as a base for an observational study but the paper does not contain any measurements about how reverse engineers perform their activities. Even though the authors did not perform any experiments, the study still deserves a special attention because it proposes an approach which is suitable to perform a quantitative study of RE.

Finally, the recent work of Votipka et al. [VRM$^+$20] is what we can consider as the first human study about RE, focusing on what high-level process reverse engineers follow and what technical approaches they adopt. The authors' goal was to improve the design of RE tools to make them more usable and intuitive. However, due to the lack of prior work outlining REs' processes and no theoretical basis for building quantitative assessments, the authors also performed a number of semi-structured interviews in which 16 participants recalled anecdotes of a binary they had reverse engineered in the past. This provides technical details about their strategy and experience, including when they switched from a tool to another, which hypothesis they formulated, and which type of documentation they consulted.

If the literature covering RE is scarce, a vast amount of work has been performed instead in the program comprehension field. Indeed, RE can be seen as a program comprehension problem applied to assembly code, with the goal of recovering the high-level abstractions needed to understand the program logic. For this reason, we collect here the most critical human studies related to program understanding. One of the leading research directions in program comprehension shows that programmers adopt non-linear ways to interpret source code, reasoning at a level of abstraction higher than the code itself [Let87, LGHM07, AS96, RTKM12, Bro83, AvML98]. A well-known model about these high-level representations is what researchers refer to as *beacons*: beacons are patterns that experienced programmers can recognize when reading the source code [Hoc14, KMCA06, Pen87]. The utility of beacons is mainly related to assessing some hypotheses that developers do about some unknown parts of the program, such as when they need to maintain some code base, as described by Littman et al. [LPLS87]. Alternatively, Gugerty [GO86] argues that developers can use debuggers to verify some behaviors within the source code they are analyzing (e.g., by checking whether a variable contains the expected value at some point of the execution). It is also worth mentioning that some of these papers study program comprehension by performing a comparison between *experts* and *novices* [GO86, FWS93, WFS93]. We believe this to be a critical factor in understanding the impact of the experience, and this methodology served as inspiration for the experiments we present in this thesis.

Finally, few studies have investigated the usability of RE tools. For instance, researchers have looked at improving the usability of decompilers [YEGPS15, JLS+18], showing that better variable naming and a reduced number of GOTOs affected positively the readability of the pseudocode. In the context of vulnerability discovery, Do et al. [DAL+17] proposed a static analysis framework that allows the developers to write code and run in parallel the static analyzer to help programmers to better manage the large number of alerts generated by the tool. In 2017, Shoshitaishvili et al. [SWD+17] showed that the communication between a fuzzing engine and non-skilled reverse engineers can increase the rate of discovered vulnerabilities by taking advantage of human intuition.

5.2 Scope of the study

As introduced in Section 2.3, RE includes a large variety of practices and approaches, depending on the context, the domain and the goal of the analysis. In this study, we focus our investigation on the core activity that is part of any binary RE process: the static code understanding as presented by interactive disassemblers.

Although this activity represents only one portion of the RE process, we believe it deserves a special attention for several reasons. First, it is particularly interesting as the low-level nature of the Assembly language forces the human mind to make an additional effort when reading the instructions. In fact, the reverser needs to understand the effects of what she reads on the machine that will execute the code as well as mentally reconstruct high-level patterns (such as loops and branch conditions) and data types.

Moreover, this approach mirrors the initial studies of the program comprehension community, where various authors initially focused on how users read source code rather than directly embedding debuggers in their experiments [Let87, LGHM07, AS96, Bro83, AvML98]. For these reasons, we decided not to include any decompilers/debuggers in our pipeline, focusing instead on an in-depth analysis of the static assembly code comprehension process.

5.3 Methodology

To conduct a detailed investigation of how humans perform a RE task, we needed to replace the interview format adopted by previous studies with a fine-grained observation of subjects' actual behavior when requested to perform different tasks related to binary reverse engineering.

While the required data could be easily collected in a lab, for instance, by using eye-tracking equipment to monitor the participant behavior [PHG+04, CG07, TFSL14, BT06], this approach would introduce several problems. First of all, skilled reverse engineers are rare and remotely-accessible experiments are required to collect enough participants with different backgrounds. Second, even simple RE exercises require hours of concentration, which is difficult to achieve while under observation in a lab (especially when the candidate needs to keep her head stable to allow for proper monitoring). Therefore, we opted for implementing a web-based platform specifically designed to conduct our experiments.

The platform needs to be capable of extracting many low-level metrics, such as how much time a person spends looking at each basic block, how she explores and navigates the binary program, and how she annotates and manipulate the assembly code (e.g., by renaming functions and variables) along the way. Moreover, the interface needs to closely resemble the interface of existing reverse engineering tools (such as IDA Pro, Ghidra, and Binary Ninja) to let the users interact with a familiar environment. Finally, the system should incorporate special techniques (such as Restricted Focus Viewer [JBM03] to blur basic blocks that are not currently selected) and a variety of instrumentations to collect a rich set of *raw* low-level information.

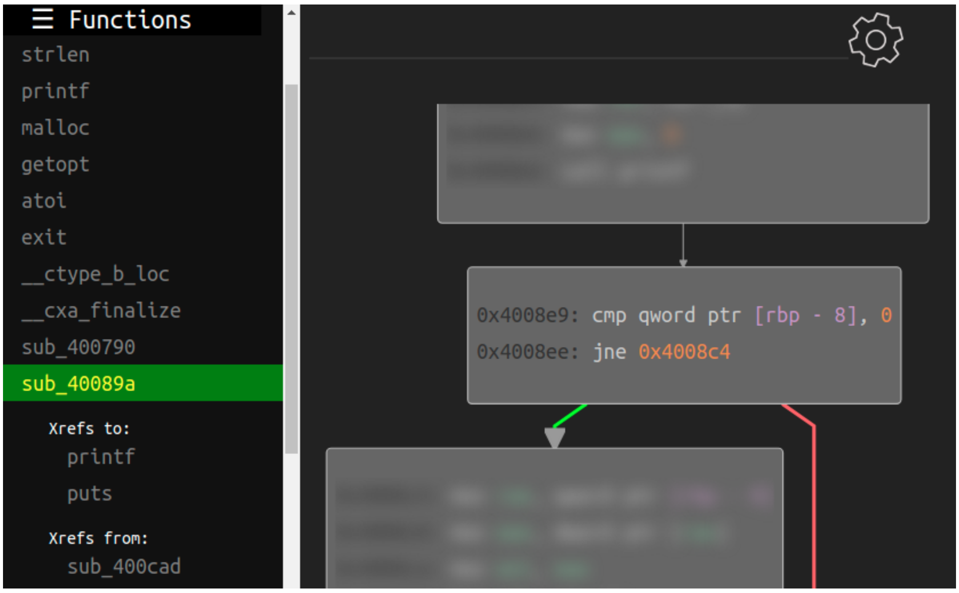
The low-level metrics extracted by our online platform act as basic blocks for the subsequent analyses and characterizations. In this second phase, we manually reviewed the collected data to identify high-level skills starting from the low-level metrics. In this respect, a significant challenge is that we did not know a priori which skills were more important than others and in which context they may become relevant for solving a reverse engineering task.

5.3.1 Online Platform

Our dedicated online platform provides users with an interface and a set of functionalities, which mimic common interactive disassemblers. The system required users to register an account to allow them to take breaks and perform different tasks at different points in time. After the registration, a "Welcome" page described the various tests and guided the participants through the system's functionalities. The user could then select one of the available tests and proceed with it.

A snapshot of the interactive RE interface is presented in Figure 5.1. The left panel shows the list of the functions that are present in the binary as well as those imported from external libraries (such as `printf`). When the user visits a function, e.g., `sub_40089a` in Figure 5.1, the system highlights it with a green bar and displays the list of code cross-references (Xrefs) below the function name. Xrefs are divided into `Xrefs-from` (functions containing a `call` instruction that transfers the control to the visited function), and `Xrefs-to` (functions called within the body of the visited function). The user can control the application by using the mouse (by panning, zooming, clicking on links, accessing a contextual menu with the right mouse button) or the keyboard (by using common shortcuts taken from IDA Pro for moving back and forward, rename variables and functions). The right panel instead is in charge of showing the Control Flow Graph (from now on CFG) of the disassembled function where we decided to resolve library calls with their symbol name and to replace strings' addresses with the string itself. Our UI also includes a call graph view, where the users can visualize the relationship between the function inside the binary.

Enabling the user to have a complete view of all basic blocks at the same time would not allow us to track her progress through the program at the required granularity. Therefore, by taking inspiration from similar experiments performed in the code comprehension community to measure the user attention [JBM03, BT04], we decided to implement a *Restricted Focus Viewer* [JBM03] (RFV) solution. This technique provides results comparable with eye-tracking methodologies by dynamically blurring parts of the screen and letting users control the visible area. In our case, only



```
0x4008e9: cmp qword ptr [rbp - 8], 0
0x4008ee: jne 0x4008c4
```

Figure 5.1: Part of the UI of our reverse engineering framework in the code navigation mode showing on the left the functions' list and on the right the CFG of the selected function

one basic block at a time is readable while all other ones are blurred. For instance, on the right side of Figure 5.1, the central basic block (from now on BB) is unblurred, whereas the ones above and below are blurred. When the user moves the mouse over a different BB, the system immediately shows its content and re-sets the previous one in a blurred state.

The main disadvantage of this solution is that it can delay the user activity and, as discussed in Section 5.5.4, it can also prevent rapid glances over different parts of the screen. Even though we are aware of the fact that RFV represents a limitation of our work (as detailed in Section 5.7), it represents the only solution to precisely measure the basic blocks observed by each participant, while enabling our experiments to be conducted online with users located in different countries.

For each action the user performs over the visible element, the framework generates an event and sends it to our backend server. These events include *New function access* (when the user clicks on a function name), *New basic block visit* (mouse over the new basic block), *Function rename* (right click or keyboard shortcut), *Variable rename* (right click or keyboard shortcut), *Comment* (right click or keyboard shortcut), *Follow the jump to an address* (double click), *Jump to address* (double click on the address), *Move* *backward* to the previous basic block (keyboard shortcut), *Follow Xrefs-to or Xrefs-from* (click on the desired xref), and *Solution submission* (click on the dedicated link). For each of the cases mentioned above, the web interface generates a JSON request containing a timestamp, the event type (i.e., the action), the position in the binary (i.e., the function address and the BB), and depending on the action type, the arguments. For example, when the humans rename a stack variable, the JSON string will contain the new proposed name and the old stack offset as further arguments.

All the events, along with the user's changes on the code (such as renamed objects, comments, and previously accessed locations), are stored in a database for further analysis.

5.3.2 Challenges Design

The main problem we encountered when designing our tests was to find a balance between the complexity of the binaries, the amount of data we could collect from them, and consequently the number of people that we could recruit.

Modeling the complexity of a RE task is not easy because a binary could include many features that make the process of understanding its internals more complex. For instance, obfuscated code would require dynamic analysis or access to the binary file for implementing a de-obfuscation algorithm, thus resulting in less data that could be collected by our platform. We also had to design our tasks to be independent from the domain of the different experts; for instance a challenge about packing would be easier for malware experts than for vulnerability researchers. We decided instead to present binaries that implement common functionalities that can be found in any domain.

That being said, there are many potential strategies to provide a measure of the complexity of our tasks. A possible way to accomplish this goal is to rely on the complexity of the source code, as done by [SKBM06] to formally describe the difficulty of their binary challenges. Peitek et al., [PAP+21] demonstrated, with the use of Functional magnetic resonance imaging (FMRI), the existence of a correlation between such source code metrics and the brain activation registered in users that perform code comprehension tasks. Therefore, we compute a total of twelve metrics (including the Halstead metrics, the cyclomatic complexity, and the number of functions and lines of code) and use these values to assess the difficulty of our assignments. All values for the two assignments are reported in Table 5.1 (in Appendix). When crafting our challenges we used these metrics of the tasks reported by Sutherland et al. [SKBM06] as a lower bound to make sure that our tasks were sufficiently complicated.

Table 5.1: Complexity metrics of the two assignments

Metric	Test 1	Test 2
Lines of code	146	207
Operators count	426	673
Distinct operators	35	38
Operands count	207	338
Distincst operands	89	87
Program length	633	1011
Program vocabulary	124	125
Volume	4402	7042
Difficulty	39	73
Effort	171678	514095
Cyclomatic complexity	14	19

After some internal experiments among the authors, we settled for three binaries. Although we understand that three binaries cannot provide a detailed view of the skills that expert reverse engineers acquired after many years of practice, we believe this choice to be a good tradeoff between the amount of data we can collect and the time each participants would need to invest in our exercises.

The first challenge binary was the smallest and only served as a warm-up to make the users comfortable with our tool's interface. Thus, we did not collect data from this first assignment. The other two binaries, which from now on, we will call TEST_1 and TEST_2, were inspired by typical reverse engineering problems in Capture the Flag (CTF) competitions. CTFs are popular games designed to challenge their participants to solve computer security problems. The goal of a RE challenge in a CTF is often to recover the input that needs to be provided to a given binary to produce a specific output. This had the advantage that solutions are small and can be easily verified on our side while still requiring the participants to "understand" the full logic of the target binary. Both the programs were written in C language and compiled for a Linux x64 machine with the `gcc` compiler.

In our tests, all binaries include a *target* function whose purpose is to print the string `'Success!!'`, and the participants were asked to submit a description of the input required by the program to print the success string. To make things more challenging, all binaries were stripped from their symbols and included several "useless" snippets of codes, which had no effect on the problem's solution.

Test 1. The first binary consists of a simple server listening on port 8888 and accepting new incoming connections. For each connection, the server would

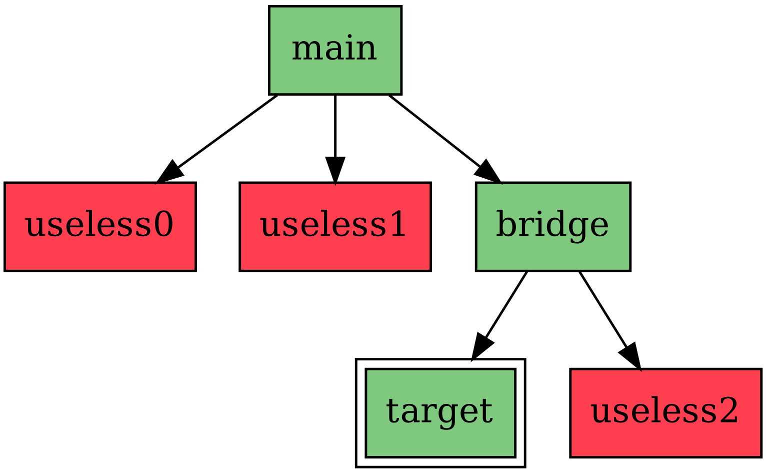


Figure 5.2: Call Graph of Test 1

I) spawn a new process (using the `fork()` function) to serve as a connection handler, II) increment a global counter, and III) invoke the target function that would print the success string if the global counter is equal to three. Therefore three clients need to connect to the server to trigger the `Success!!` string.

The challenge requires the participants to recognize the assembly patterns associated to simple network actions (e.g., the initialization of the socket structures and `bind()`, `listen()`, `accept()` APIs), and the parent/child relationship during a `fork()`. For the sake of clarity, we sketch the call graph of the binary in Figure 5.2. The figure shows in green the three functions that need to be reversed to solve the exercise, and in red, the three additional functions that play no role in the solution. Two out of the three additional functions are responsible for handling error conditions generated along the binary. The purpose of such procedures is to assess if participants can easily recognize and ignore functions that only generate error messages. The third procedure is the one that implements the connection management.

Test 2. The second binary implements a simple list management application. The application accepts two parameters, a list of integer numbers,

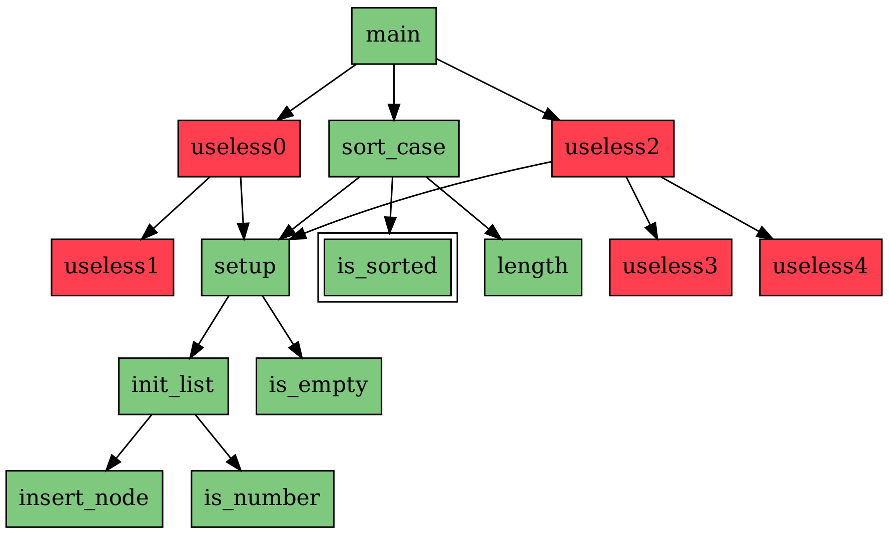


Figure 5.3: Call Graph of Test 2

and one letter that specifies the required operation: (`a`) – the application sums the elements of the list and prints back the result; (`r`) – the application prints the list in a reversed order; (`s`) – the application checks whether the list is sorted and contains at least four elements. If both conditions are satisfied, the program prints the success string.

This second binary is more complicated than the previous one, and all operations are performed over linked lists of custom data structures. To ensure that the difficulty was higher than TEST_1, we verified that all twelve complexity metrics had higher values than in the previous test. The challenge requires the participants to be familiar with linked lists in assemblers (i.e., on the way C `structs` and pointers are compiled in binaries) and to recognize list-related operations (including a bubble-sort implementation). Figure 5.3 represents the simplified version of the call graph: as in the previous case, we label as *useless* the functions that are not related to the challenge solution. For all the other functions, we report their self-explanatory name. However, the symbols' names were stripped from the binaries, so the participants did not have this information.

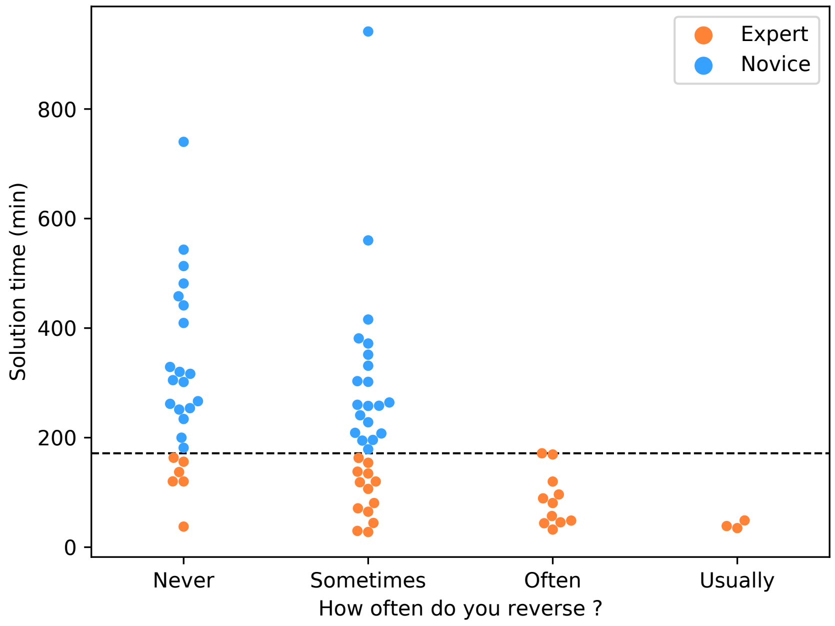


Figure 5.4: Relationships between how often the subject reverse binaries and the total time spent to solve the exercises.

5.4 Participants recruitment

We ensured that all methods and experiments performed for this work are in line with our institutions' research ethics guidelines and our country regulations on data collection and retention. The participants were recruited over a period of several months and the invitation was sent from our institutional email address as proof of credibility. The text, reported in Appendix A.1, contained a complete description of the experiment with the link to our online infrastructure. As we specify in the recruitment email, we did not provide a compensation for our experiments and we only collected anonymous data.

In particular, we contacted *students* who took a binary analysis or reverse engineering course in three different universities. All students had been previously trained to reverse binary programs, but while some were still beginners, others already had experience by playing CTF competitions. We also contacted nine different top CTF teams, asking for *players* who usually solve complex RE challenges to participate in our experiment. Overall, 95 users responded to our request, but only 72 completed successfully the two tests.

In order to compare the effectiveness of different approaches to read the disassembled code, we split our participants into two groups: *experts* and *novices*. On the one hand, simply relying on the "reputation" of the participants could lead to biased results in our data analysis. On the other hand, self-evaluation questions can also produce biased results because humans tend to adjust their answers depending on their concerns with the interviewer's perception [TY07, HGK03]. Therefore, we decided not to divide the participants in two a priori groups, but rather to combine their self-reported experience with the time required to solve the tasks. First, when visiting the website, the participants were asked how often they reverse engineer binary code on a four-point scale in ascending order of frequency: *never*, *sometimes*, *often*, and *usually*. Then, once all experiments had been completed, we identified the time required by the "worst" participant who reported to reverse binaries *often* or *usually* (i.e., 172 minutes). Finally we adopted this value as a threshold: participants who took less time than this threshold are considered experts, otherwise novices. The two groups contained respectively 33 experts and 39 novices. It is worth noting that all CTF players ended up in the expert group.

Figure 5.4 shows the relationship between the answer to the frequency question and the time required to complete the two assignments for the two classes of users whereas the dashed horizontal line represents the threshold we have inferred.

5.5 Data Analysis

We now discuss the results of the participants who completed the two exercises (39 novices and 33 experts). First of all, as we expected, novices and experts spent a different amount of time to complete the assignments. In fact, the two exercises combined took between 24 and 172 minutes (92 on average) for the subjects in the experts' group and between 178 and 941 minutes (340 on average) for novices. In other words, even though the exercises were relatively simple, beginners were, on average, 3.7 times slower than experts, and the fastest beginner was 7.4 times slower than the fastest expert.

To avoid bias, we computed the confidence intervals for the two groups of users, with a confidence value equal to 0.95. In this second scenario we obtained that the time required was between 75 and 110 minutes for experts

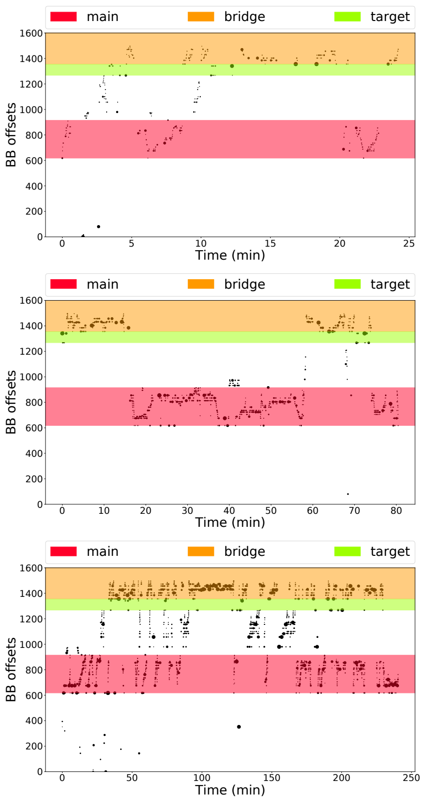


Figure 5.5: Three distinct RE sessions of Test 1 showing the time spent on each basic block during the session

whereas it ranged from 289 to 391 for novices.

Moreover the application of a 2-sample t-test over the two groups in relation with the solution time confirmed us that it is meaningful to separate the two groups in terms of time needed to accomplish the task (t-test 9.31, p-value 3.5e-10).

We also analyzed the solution time by splitting the users according to their answer to the initial question about how often they reverse binaries.

As shown in Figure 5.4, it took on average 301 minutes for users who answered 1 (rarely reverse binaries), 233 for those who answered 2, 86 minutes for those who answered 3, and finally 40 minutes for the only three experts who reported to reverse binaries on a daily basis. This shows that while all participants in our expert group were fast, on average, those who perform this task more often tend to be faster.

Mining for Strategies

We started our analysis by manually inspecting the telemetry data collected from the users' sessions, looking for macro-differences that could indicate the use of different strategies. As an example, Figures 5.5 shows a graphic representation of the behavior of three users during the first exercise (time is on the X axis and BB addresses on the Y). The horizontal bands of different colors represent the three useful functions (those required to solve the exercise), while the white region indicates the BBs located in other irrelevant parts of the program. Each dot corresponds to the user focusing on a given basic block for a certain amount of time (expressed by the size of the circle). The labels `target` and `bridge` respectively indicate the function that prints the success string and the function that has `main` as the caller and `target` as one of the callees.

The first two graphs belong to experts (the fastest in our test and an average one), while the bottom depicts a beginner session.

The three graphs clearly show very different approaches to reverse the same binary. The second expert spent a considerable amount of time on `main` (the red band), while the first moved away from it after a few minutes and returned to its code only after a first overview of the binary. Moreover, the order of their visits is different. The first started from `main` while the second user started the exploration from the `target` function (where the success string is printed). However, even if the first approach is more efficient, the first expert spent more time looking at unrelated code (dots in the white band) than the second (9 against 4 minutes).

The novice session appears more chaotic. It contains many more points (i.e., BB visits), reaching a total of 3469 visited blocks, and the user kept switching back and forth between the three main functions, probably trying to make sense of the entire program.

Looking at all 72 graphs, it seems like everyone has their own style. However, we are interested in generalizing these first observations and finding whether the strategies adopted by experts have something in common that does not appear in the novice sessions. We also notice considerable variance among the experts themselves, so we want to study possible differences among users in the same group.

To perform this analysis, we first distilled the collected low-level events into several high-level features representing observable behaviors that we could identify in our dataset of participants. We then tested whether each feature was substantially different between experts and novices and whether it was positively correlated to the overall solution time. While this second aspect does not necessarily imply causation, it can still show which set of

Table 5.2: Prevalence of Function-level Strategies for novices and experts while approaching the two binaries

Strategy	**Test 1**		**Test 2**	
	Novices	**Experts**	**Novices**	**Experts**
Sequential	4	-	8	-
Backward	2	6	5	8
Forward	33	27	26	25
Depth-First	0	2	1	6
Breadth-First	11	8	16	12
Hybrid	28	23	22	15

techniques are more commonly used by those reversers who could complete the exercise in a shorter amount of time. Before performing the statistical test, we checked if the data distribution (especially for time samples) was normal and, in negative case, we applied a log-transformation to normalize it. For each test that we executed, we collected the resulting p-values inside a vector, and we used the Bonferroni method to correct them with an input alfa of 0.05 (all additional hypotheses we tested are listed in Appendix 5.7). The corrected alfa that we obtained is 1.2e-03 and all values that we report in the chapter already take into account the Bonferroni correction.

5.5.1 Functions Exploration

Function exploration strategies play an important role to discover the path between the `main` and the `target` functions. Once this path has been unveiled, users can focus on the BBs that compose the functions in this path and therefore they abandon their function-level strategy and drive the exploration according to what they found. For example, if we consider the second expert of Figure 5.5, we can note that she adopts a backward approach, starting from the `target` and then reaching the `main` function. Then, she focuses with more attention on the BBs of such (and other) functions to figure out how to craft the proper input to solve the challenge.

Three different ways exist to move across functions: by following Xref, by direct access (i.e., by clicking on the function name in the sidebar), and by following the CFG (i.e., by clicking on the `call` instructions or by using the `ESC` key to step backward). Accordingly, we identified three main exploration strategies: *forward* (starting from the program's `main` and following the CFG), *backward* (by first searching the API call that prints the SUCCESS string and then backtracking the analysis by following `Xref` references), and *sequential* (i.e., by exploring each function independently of its role or position in the callgraph).

Whenever a user explores the code of a function and encounters a `call` instruction, she can decide to proceed either *depth-first* or *breadth-first*. In the first case, the reverser visits each function vertically until she reaches a leaf. In the other, she explores the called functions horizontally before moving deep into each part of the call graph. To discern between the two strategies we cannot use standard DF and BF detection algorithms, as users often alternate between the two methods. Therefore, we considered a sliding window of two visits on the call graph and compared consecutive bi-grams by looking for typical BF or DF patterns. For instance, a typical window of a user using a DF approach consists of two visits to different functions following the direction of the graph's edges. On the other hand, the bigrams of a BF strategy contain consecutive bigrams of the same two functions, but appearing in alternate directions (e.g., $f - g$ followed by $g - f$).

We say that a participant predominantly uses a given strategy if it employs it at least 50% more frequently than the other. When this does not happen, we assign the user to a *hybrid* category, which means that the reverser adopted both exploration strategies at different points in time without a clear preference. The prevalence of the different exploration techniques is summarized in Table 5.2 for both novices and experts, and it shows that experts and novices clearly use different techniques. The sequential exploration is adopted by a non-negligible amount of the beginners (4 in the first test and 8 in the second one), but none of the more experienced reversers follow this approach. Users in both categories prefer the forward rather than the backward exploration. We can also see that BF visits are much more common than DF, and it is important to note that almost none of the novices resorted to a DF approach.

So far, we learned that experts tend to use different strategies, but it is still unclear whether a given strategy impacts the time required to solve the exercises. We performed an ANOVA test by splitting the participants' solution time into 3 groups (*depth-first*,*breadth-first* or *hybrid*), and applying the `one way` function to these. We ran a separated test for each challenge because some participants changed their strategy depending on the task, but all tests failed (p-values for each challenge were 0.17, 0.19 with effect sizes of 1.8 and 1.6 for the *forward-backward-sequential* classification and 0.14, 0.2, effect sizes of 2.1 and 1.9 according to the *depth-breadth* separation). In fact, as depicted in Figure 5.6, all techniques were used to efficiently solve the two exercises.

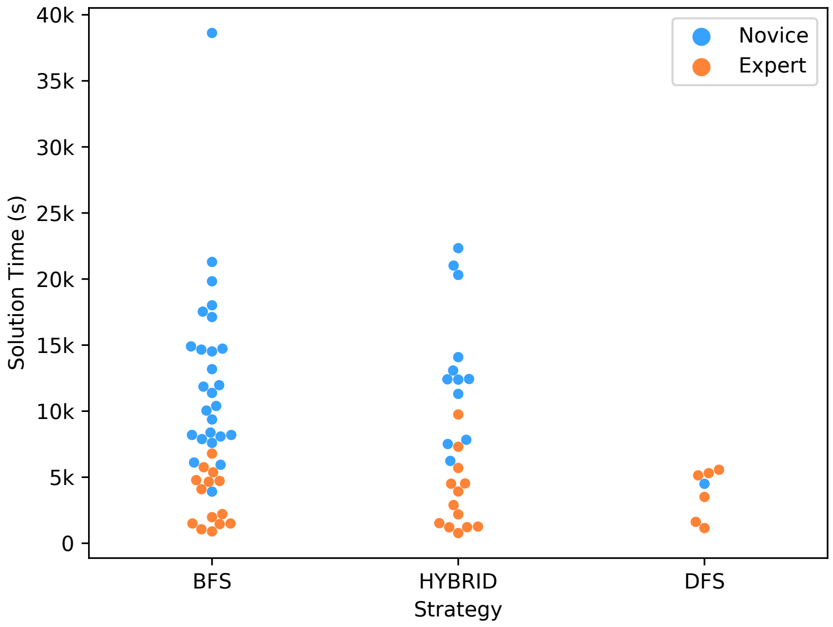


Figure 5.6: Time needed to solve Test 2 grouped by strategies.

5.5.2 Code Selection

We now check *where* the reversers spent most of their analysis time. Table 5.3 shows all functions in the second binary, and for each of them, it reports several metrics. The table is divided into two parts: the top half lists useful functions, i.e., those involved in the solution of the problem. The bottom half lists instead the five 'useless' functions (the binary accepts three different commands, but only one is required to print the success string). However, since also the related functions include irrelevant paths (e.g., to handle error conditions), in the first two columns we report the total number of basic blocks in the function and the total number of 'good' blocks (B_{good}), which are those that must be reversed to conclude the exercise. The table also reports how much time (both the absolute median time and in percentage over their entire session) experts and novices spent on each function and the overall ratio between the experts and the novice time (last column) computed as the absolute median time of novices divided by the absolute median time of the experts for that function.

There are two interesting observations we can make from these results.

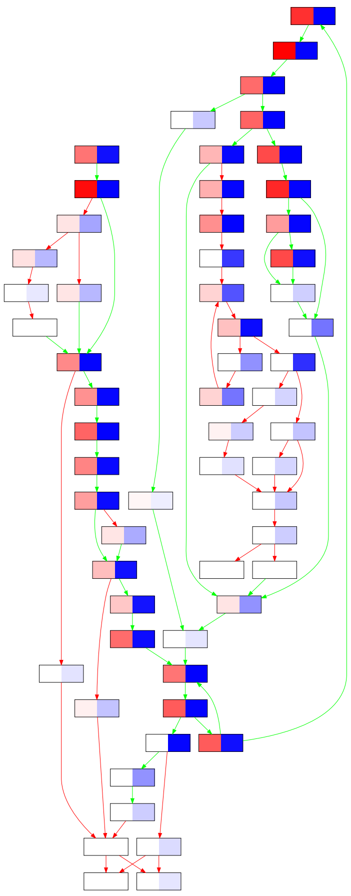


Figure 5.7: Average times spent in the BBs of Test 1

First of all, all participants spent most of their time on *main* (because it was longer) and on the functions that operate on linked lists. However, beginners were impacted more by the nature and complexity of the function. For instance, they spent much more time (4.8x slower than experts) to recognize that `is_number` only verifies that all parameters are integer numbers. We believe that this is due to the fact that similar simple functionalities are encountered frequently by reversers and therefore are easily recognized by experts.

Nevertheless, the most striking result is the fact that, in percentage, novices spent almost the **same** percentage of their time (8.6% vs 8.3% for experts) on reversing useless code (even if in absolute terms they still spent

Table 5.3: Median Time Per Functions for task 2

Function	**BB**	BB_{good}	**Experts min (%)**	**Novices min (%)**	**Time Ratio**
main	16	12	9.1 (15.8%)	29.4 (13.9%)	x3.2
sort_case	8	6	5.7 (9.6%)	18.2 (8.6%)	x3.1
setup	6	4	4.4 (7.8%)	13.4 (6.3%)	x3.0
is_sorted	12	10	10.4 (18.1%)	28.9 (13.7%)	x2.7
init_list	8	7	8.7 (15.1%)	38.7 (18.3%)	x4.4
is_empty	1	1	0.26 (0.4%)	1.0 (0.5%)	x3.9
insert_node	7	6	5.8 (10.2%)	28.0 (13.4%)	x4.8
is_number	9	8	4.7 (8.2%)	22.9 (10.9%)	x4.8
length	4	4	3.5 (6.2%)	12.0 (6.0%)	x3.5
useless-0	6	0	1.0 (1.8%)	2.8 (1.3%)	x2.8
useless-1	4	0	0.5 (0.9%)	2.9 (1.4%)	x5.7
useless-2	7	0	0.9 (1.6%)	2.9 (1.4%)	x3.1
useless-3	4	0	1.5 (2.6%)	5.4 (2.6%)	x3.6
useless-4	4	0	0.8 (1.4%)	3.9 (1.9%)	x4.7
TOTAL	96	58	57.2 (100%)	210.4 (100%)	x3.6

four times more than experts). At first, this seemed counter-intuitive. In fact, we expected experts to be better at quickly skimming through the code and ignoring it if it was not related to their task. However, given the numbers in Table 5.3, we hypothesized that this discrepancy is because novices were so slow to understand the difficult parts of the code that, in percentage, they appeared faster in discarding the non relevant ones. We computed the same values for the first binary and we observed the same trends even if in that case the number of functions is minor compared to the second challenge (only 6). Indeed, the *main* is still the function where users spent most of their time and the effort dedicated to the useless functions is basically the same in percentage (13.1% for experts vs. 12.5% for novices). Table 5.4 the values for the first challenge.

Hence, we decided to measure the total number of basic blocks that were visited by each participant. In total, the two exercises combined contained 155 basic blocks, but only 94 (61%) of them were actually along the solution path. To complete the two exercises, the median expert completely skipped (i.e., never even checked once) 24 basic blocks, while the median novice skips only 6 of them. Indeed, this fact shows that experts could cut entire branches (or functions) by only looking at a few of their blocks.

For instance, Fig. 5.7 shows the CFG of Test 1. The green edges point to interesting BBs while the red ones point to useless BBs. Each node is split in half: the intensity of the left side represents the amount of time

Table 5.4: Median Time Per Functions for Task 1

Function	**BB**	BB_{good}	**Experts mins (%)**	**Novices mins (%)**	**Time Ratio Time Ratio**
main	23	17	14.6 (44.6%)	65.3 (45.4%)	x4.4
bridge	12	9	11.4 (34.9%)	52.3 (36.3%)	x4.5
target	3	3	2.4 (6.3%)	8.2 (5.7%)	x3.4
useless-0	1	0	0.17 (0.5%)	0.60 (0.42%)	x3.5
useless-1	4	0	0.58 (1.8%)	2.07 (1.4%)	x3.5
useless-2	16	0	3.5 (10.8%)	15.4 (10.7%)	x4.4
TOTAL	59	29	32.6 (100%)	143.8 (100%)	x4.4

spent on that BB by the experts (on average); the right side represents the same for novices. If we consider the noninteresting paths, the blue intensity is generally higher than the red. Experts mostly recognize that some code parts lead to useless BBs by just reading the first BBs of that function and then recovering the correct path to the target function. Novices instead needed to go through also the noninteresting parts of code before understanding that they do not need them for their purposes.

Finally, we performed a 2-sample t-test using as an hypothesis the correlation between the group (i.e., expert/novice) and the time spent on non-useful portions of code. With a p-value of 5.3e-04 and a t-test of 4.86 we can conclude that indeed there are statistically significant differences in the way the two groups of participants look at the non-interesting parts of the binary.

5.5.3 Birdseye Overview

Experiments on code comprehension conducted by Uwano et al. [UNMM06], and independently validated by [SFM12], have found that users often perform an initial scan of the entire codebase to get a general idea of what the program is supposed to do. During this initial scan, the authors found that programmers went through 70% of the code in the first 30% of their analysis.

By looking at the reverse engineering sessions we collected in our experiments, we can clearly identify some reversers performing such preliminary scans. However, this behavior is not as typical as one might expect. In fact, in our data, only 36.0% of the experts visited 70% of the code blocks in the first 30% of their time. On average, at the 30% mark, expert reversers had visited only 48.2% of all BBs. The number increases to 53.4% (still well below the 70% threshold) if we only count the good basic blocks and ignore

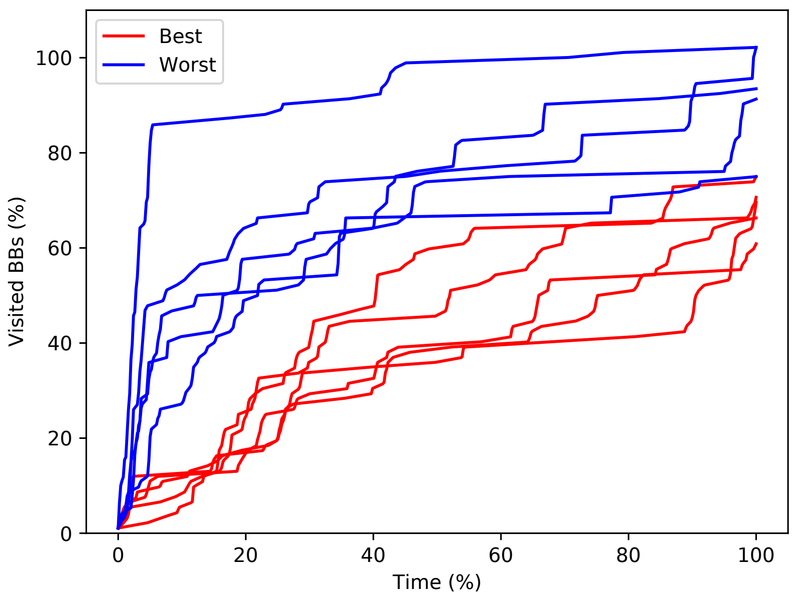


Figure 5.8: Progression of Top5 and Bottom5 experts in the second challenge.

those that were not relevant for the task. Beginners tended instead to move through each BB much quicker at first and to return back multiple times during their sessions to read again the code (we will analyze this aspect in Section 5.5.4). As a result, 69.4% of them met the 70% threshold at the 30% mark.

But there is more. Figure 5.8 shows the Cumulative Distribution Function (CDF) of the visited BB over time by comparing the top five experts (based on their solution time) against the bottom five. It is interesting to observe that the fastest reversers (in red) progressed more linearly and did not employ any initial survey strategy.[2]

This seems to suggest that a preliminary overview of the entire binary might be useful to get an orientation in large codebases, but it might not be very useful in smaller exercises. Even more surprising, we found that the majority of experts did not even 'try' to quickly skim through the code of the various functions, even though they did not know in advance anything

[2]This trend does not change if, instead of basic blocks, we perform the measurement at a function granularity (we omit the graph for space reason).

about the complexity of the task.

Figure 5.8 also confirms what we found in the previous section, i.e., that all best reversers were not fastest only because they could read and understand the code faster, but also because they reversed **less** code. On the far right of the CDFs we can see that the red curves terminate between 60% and 80% of the total BBs (remember that only 61% were along the solution path), while the blue lines fall in the range between 80% and 100%.

5.5.4 Basic Blocks Exploration

After looking at the function granularity, we now focus our attention on individual basic blocks.

Thanks to the use of the *restricted focus viewer*, our reverse engineering platform can accurately track the time spent by each participant on each individual BB. However, not all these time events are equally important. For instance, it can occur that when moving the mouse pointer between two BBs, the user accidentally moves the mouse over an intermediate BB without being really interested in its content. Our infrastructure would capture this behavior, generating an event for all three BBs. To remove the noise introduced by these spurious events, we decided to conservatively discard all the views with a duration below 500 milliseconds. This threshold is based on the fact that, according to Rayner et al. [RJP08], while reading text, the eyes stay upon each single location from 100 ms to over 500 ms. Given the fact that a BB is often composed by multiple lines, this threshold ensures that a participant had time to focus on at least one location in the BB. Anything below that would not provide much information to the reverser.

It is essential to understand that the time a reverser spends on a single BB is affected by multiple factors, including the BB complexity, the user assembly reading skills, the role of the block inside the binary, the navigation strategy of the user, and the state of the ongoing RE session. We will try to break down these factors in the rest of the section.

To begin with, for each BB we identify three different time values. First, the time each user spent on the block the first time she encountered it (T_{first}). Second, the total cumulative time (T_{tot}) each user spent on the BB over the entire exercise. And finally the longest consecutive time each user spent on the block (T_{max}).

By comparing these three time intervals, we can make several interesting observations. Figure 5.9 shows the distribution of the median time spent by each user over all the basic blocks of the two exercises. It is interesting to note how, the first time they encounter a new basic block, both experts and

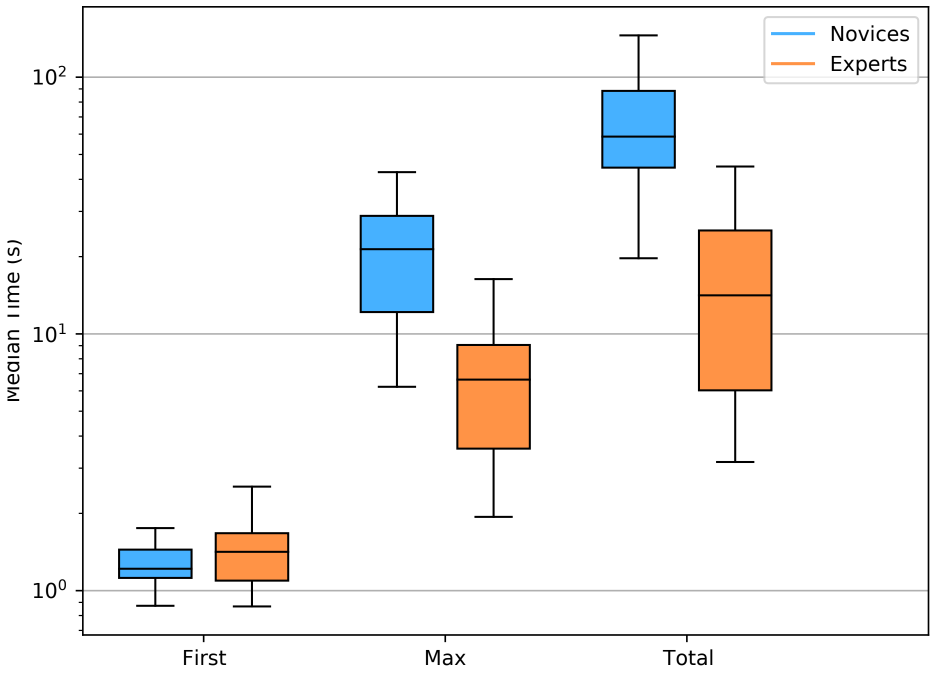


Figure 5.9: Comparison of the distribution of T_{First}, T_{Max}, and T_{Total} time among the users in the two groups.

novices spend only a few seconds on its code: on average 1.3s for beginners and 1.5s for experts. Instead, the maximum and total time spent on the blocks are over one order of magnitude higher, often lasting for tens of seconds (6.8s vs 21.9s for T_{max}, and 16.3s vs 73.4s if we compute the median times for the T_{tot}). As a confirmation of this aspect, we ran the 2-sample t-test over the values of T_{first}, T_{max} and T_{tot} collected over each user and then separated by novices and experts. Indeed, we obtained that the difference for the first visit (T_{first}) is not statistically significant (p=0.2) but the time difference on T_{tot} and T_{max} are (respectively with p=7.4e-07 and p=1.5e-08).

At first, one might easily dismiss the role of these first short visits, nothing more than a quick glance at a block while the user rapidly moved the mouse over it. It might seem obvious that the 'real' reverse engineering is performed over the subsequent visits. However, if we compute the fraction of BB that a user visited only once we see that things are more complex. On average, experts visit 28% of the BBs only once. In 80% of these cases, the visit lasted less than two seconds. This means that experts dismiss almost

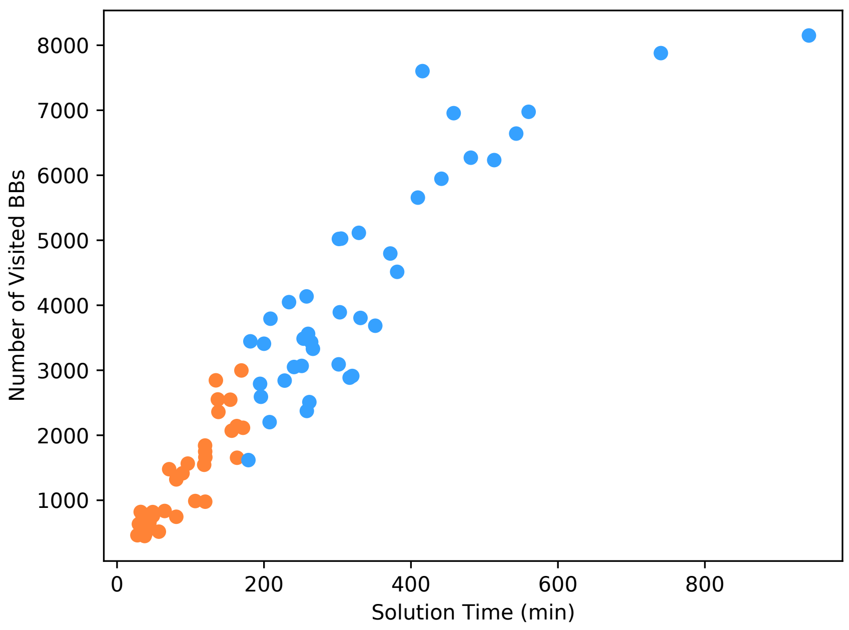


Figure 5.10: Solution time w.r.t. number of visited BBs.

22% of the basic blocks in a single glance. On the contrary, inexperienced users make a single visit only for 10% of the BB, and in total, dismiss only 7% in less than two seconds.

All the remaining BBs are visited by each reverser multiple times. In fact, even if the two programs combined contained only 155 BBs, to complete the two exercises, experts visited, on average, 1368 basic blocks and novices 4326 (2-sample t-test=9.7 and p=6.8e-12). Figure 5.10 shows the relationship between the time required to solve the challenges and the number of visited blocks.

However, visiting a block multiple times is not always a sign of inefficiency, and in some instances it is even unavoidable (e.g., those blocks that contain a function call are often re-visited when the user moves out of the function and back to the callee). We ran the Pearson correlation to test if a relationship exists between the number of times the users go through an already visited BB and the overall solution time and obtained a result of 0.68 (p-value 1.2e-05) for experts and 0.46 (p-value 2.5e-04) for novices. Therefore we investigated this aspect in more detail and computed the num-

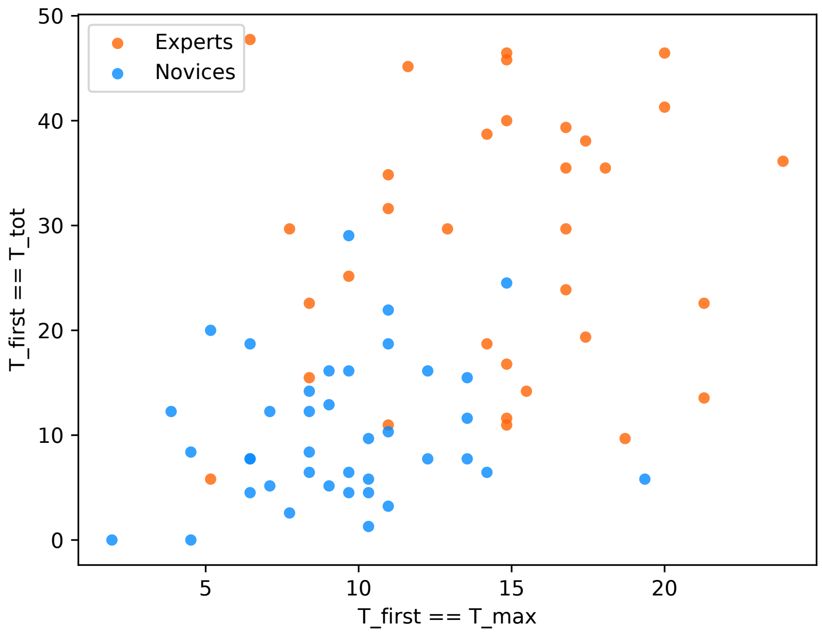


Figure 5.11: Percentage of Basic Blocks visited only once, or analyzed on the first visit.

ber of times the first visit to a basic block was not the only one, but it was the longest (i.e., $T_{first} == T_{max}$).

If we assume the most prolonged visit is when the user actually completely reversed the BB code, we can use this indicator to know whether this is performed for the first time the reverser encounters a new code. In this case, the median is 9.6% of the BB for beginners and 14.8% for experts. Again, it seems that experts tend to fully understand the code the first time they read it, while beginners go back multiple times, and in 80.6% of the cases, their first visit is not the one where they reason the longer on the code.

Finally, it is interesting to test if these short first visits are just a consequence of the fact that a reverser might be simply faster at processing assembly code. In other words, we wanted to test whether those users that have shorter first-time visits (T_{first}) also spend less time overall on the BBs (T_{tot}). However, the Pearson correlation of the 2-time values is -1.2, p-value=0.5, showing no statistically significant correlation.

Table 5.5: Correlations between visits duration and BB length

	Experts		Novices	
Hypothesis	**Pearson**	**p-value**	**Pearson**	**p-value**
T_{max} and $len(BB)$	0.29	1.0e-04	0.31	5.8e-04
T_{tot} and $len(BB)$	0.30	8.28e-04	0.33	1.6e-04
T_{first} and $len(BB)$	0.37	1.9e-05	0.37	1.3e-06

Figure 5.11 shows how a scatter plot of the two aforementioned metrics (percentage of blocks visited only once, and for which the first visit is the longest) can clearly separate the majority of the experts from novices reversers.

5.5.5 Speed Factors

In our final analysis of the different reversers' speed, we look at which factors affected the time spent on individual basic blocks.

For this purpose, we limit our analysis to those blocks that actually needed to be understood in the first place. Thus, we first remove those BB that are NOT related to the solution of the exercise as well as all headers and footers of the functions (as it might not always be required to analyze their behavior carefully). The remaining (which we will refer to as BB_{core}, and that account for 47% of all blocks in the two assignments) capture the code each user *had to reverse* to reach the correct solution.

The first hypothesis that we wanted to formulate was to study the potential correlation between the time spent on each block and the size of the block itself. Indeed, we observed that the first, total and max times are positively correlated to the number of assembly instructions contained in the basic block. However, the exciting result is that the Pearson correlations are quite small for T_{tot} and T_{max} while they exhibit an higher value for T_{first} as reported in Table 5.5. Moreover, under the same hypothesis, the correlations are always more elevated for novices.

One way to interpret these results is that the amount of time spent by reversers on a basic block is only marginally influenced by the time required to actually read (or *'parse'*) each assembly instruction. The impact is more visible for inexperienced reversers (who probably spend more time reading the assembly) and less on experienced users. To understand which other factors contributed to the reversing time, we extracted the top 5% of the basic blocks in which each user spent most of her time. Then we compared all sets to identify those blocks that were problematic for a large percentage

Table 5.6: Statistical tests w.r.t. the branch selection data (upper part) and the semantical elements data (lower part)

Hyphothesis	Result	p-value
Novices true branch & solution time	0.21	0.06
Novices close branch & solution time	0.13	0.2
Experts true branch & solution time	0.42	0.7
Experts close branch & solution time	0.87	0.4
2-sample T-test Comments	0.4	0.6
2-sample T-test Variable Renames	0.8	0.4
2-sample T-test Function Renames	0.7	0.4

of users.

If we look at total or max time, both experts and beginners spent most of their time (respectively 19% and 18% on average) on blocks that prepared the function call parameters. While usually straightforward to reverse, all reversers probably paused to reason about which values were passed to the function's parameters. If we look instead at the blocks that frequently appear among beginners but not among experts, we find a total of 20 BBs that are shared between a minimum of 3 and a maximum of 11 novices and that are responsible for an additional 9% of time on average overall. We analyzed them to unveil the assembly language (ASM) patterns that slowed down the novices while reading them. In total 6 blocks contain uncommon instructions (such as `setnz`, `imul`, and `sar`) and 7 include instructions that operate with in-memory data structures, thus requiring to reason about the memory layout of the program in that specific moment (e.g., instructions that access the i-th element of the list of Task 2). We also found 3 BBs that operate on the static strings contained in the binary. Among these 20 BBs only 4 of them have a number of instructions major than 10 while the other 16 contains less than 6 instructions (and in 10 cases they were just 3 instructions long). This finally shows that the nature of the instructions is more relevant than their number to explain the comprehension time for beginners.

5.5.6 Other Aspects

In the previous sections, we discuss several aspects we believe can capture subtle but essential characteristics of the behavior of either experienced users or beginners. We also tested many other hypotheses and tried to isolate other behaviors (reported in Table 5.7) but for which we could not find any statistical difference among our users. For these hypotheses that did not find

Table 5.7: Additional hypothesis (not discussed elsewhere)

Hyphothesis	**p-value exp**	**p-value nov**
First quartile of time spent on a BB and BB length	0.1	0.6
Interquartile of time spent on a BB and BB length	0.06	0.07
Average of time spent on a BB and BB length	0.08	0.1
Mode of time spent on a BB and BB length	0.4	0.7
T_{first} and T_{max}	0.3	0.2
T_{first} and T_{tot}	0.8	0.8
Solution time and number of BBs she skimmed (i.e., she did a quick look at BB and then a longer one to the same BB)	0.4	0.5
Solution time and how many times the user went back to the previous BB instead of going forward	0.1	0.2
Glanges (i.e., visits of less than 2 seconds) and BB length	0.1	0.1
Solution time and how many times she went to a true branch	0.8	0.1
Solution time and how many times she went to a close branch	0.3	0.2

a statistical validation we report the p-value that we obtained after running the Pearson correlation, but we omitted the correlation value itself for space reasons, since it was not meaningful. However, we want to add two more short points to our analysis regarding the impact of the user interface in branch selection and the other events we collected from our platform.

Branch Selection - when visiting a conditional BB for the first time, beginners choose to explore the true branch first in the 41% of the cases, whereas experts followed the true branch in the 42%. However, we found that the physical position (on screen) of the basic block is much more important than its logical one. In fact, our results show that both experts and novices tend to simply visit first the closer basic block, respectively in 87% and 88% of the cases they encounter a branch. Finally we tested the hypothesis that the choice of either true branch or close branch as a next step has an effect on the overall time to reverse engineer the binary. However for both experts and novices we obtained p-value > 0.1 (values are reported in Table 5.6).

Comments and Rename Actions - we also investigated the use of the other features implemented in our infrastructure: comments, variable renames, and function renames. On average, we recorded 24 comments among all the expert sessions, whereas we count only 11 from novices. The same trend happens for variable renames (19 vs. 7) and function renames (12 vs. 2). One more time we applied the 2-sample t-test for each of the semantical elements created by the user, divided for experts and novices. The results of the test (reported in Table 5.6) show no statistical significant relationships between these features and the users performance. At a first look, this result looks like surprising as we would expect that a statistical significancy exists between the usage of semantical elements, the solution time and the experience level. However our hypothesis for this behavior is that probably the statistical relationship between the use of semantical text fragments and the RE performances become more and more evident while observing this on larger and more complicated codebases (potentially together with other reversers with the same experience and working in the same team). We will discuss more carefully about challenge design limitations in Section 5.7.

5.6 Summary of Findings

In this study we quantitatively measured the behavior of 72 reversers, both experts and novices, over a total of 272 hours of RE activity. By looking past the individual features discussed in the previous section, we will now summarize the main findings that emerged from all our results.

First of all, we found that each user is unique and has her strategy and her way to reverse binary code. However, by looking under the apparent diversity of actions, we can identify a number of core strategies. To begin with, novices move prevalently forward from the program's `main` while experts mix forward and backward movements. While statistically the difference is clear, there are notable exceptions in all groups, showing that one can be very efficient independently from the strategy it adopts (except for the sequential scan that is only used by the very beginners).

Experts also exhibit a more linear progress, avoiding to jump back and forth among the same basic blocks they already visited in the past. Moreover, they make every visit count, even the first one. This allow them to dismiss 22% of the basic blocks in a single observation, which often last less than two seconds. The 70-30 birdseye scan observed several times in studies of program comprehension does not seem to apply to binary reversing, at least at the small scale dictated by our exercises. Instead, the experts' ability to quickly identify and *ignore* the regions that were not relevant for their task was one of the essential aspects that distinguished experienced users from beginners. This, which fits the self-reported techniques that Votipka et al. [VRM+20] group under the name of *subcomponent scanning* could, in fact, be related to the ability of the expert's brain to recognize code patterns, but more focused experiments (e.g., with brain EEG sensors) are needed to investigate further and validate this hypothesis.

Finally, our experiments show that the number of instructions is a very poor predictor of the time required to understand a piece of code and that the presence of less common instructions has a more noticeable impact only on novices.

5.7 Limitations

When we designed our experiments, we had to make many choices to balance the difficulty of the problems (and, therefore, the time required for completing the exercises), the amount of data we could collect, and the impact of our instrumentation on the user experience. These choices might have introduced biases in the results or might have prevented us from observing some aspects of the users behavior.

Expertise - In our study, we measure the expertise of a user in three ways. First, based on "reputation", i.e., by inviting as experts only those users that can already solve very difficult reverse engineering challenges. Second, by the frequency on which each user reverses binary files (as reported in the questionnaire during registration), and finally by the total time required to solve the two assignments. However, one may argue that a good reverser does not necessarily need to be fast—but some may prefer instead to be meticulous and precise in her findings. New experiments should be designed only for expert reversers to measure this aspect by providing them with more challenging assignments where precision may be more important than speed.

Restricted Focus Viewer - The use of a RFV to capture the part of the code a user is currently focusing on is a standard methodology in comprehension experiments. While it allows for remote participation without the burden of on-site (and uncomfortable) eye-tracking solutions, this choice also introduces some limitations. First, it impacts each participant's overall speed. It also prevents glances, in which users quickly look at a different basic block, maybe just to check a register or the final instruction. In our settings, this requires moving the mouse, and therefore users might perform this task less often than in an unconstrained environment. We can also hypothesize that the issue with the glances affects the order in which basic blocks are visited. Another potential drawback is that it could technically discourage the participants from using the birdseye overview (Section 5.5.3), forcing them to rely mostly on their own memory to remember a previously visited basic block. However, this affects only a reduced number of cases: moving the mouse back to a previous basic block is “expensive” only if we want to quickly recall a specific location of that block (e.g., a register, a single ASM line) as in the case of glances, but it becomes fundamental, therefore justifying the time “expense,” if the participant wants to entirely read the BB.

These factors can affect the code comprehension process by distorting the way it is performed. In absence of RFV, we could expect a higher number of glances and therefore a shorter time to discard some blocks of code. Unfortunately the only way to determine how the RFV influences our findings would be to compare it with some data collected using the same tool without RFV, which is impossible by design. Thus we can only acknowledge this limitation and hope that future studies will be able to overcome it with different technologies or with a different experiments' design (e.g., smaller group of experts monitored with eye tracking devices).

Nature/Number of the Exercises - It is possible that the tasks we ask the participants to perform may affect the ecological validity of the behaviors we observed in their session. In particular, more difficult problems and larger codebases could require different strategies or help identify other aspects that differentiate one expert from the others. However, in this measurement study, we wanted to include beginners and, therefore, opted for tasks that could be solved (even if with more significant effort) by non experienced reversers. While the number of tasks could be extended , this would increase the time to complete our assignment, especially for some participants who already spent several hours with the current configuration (and that are not affiliated to our group). Even if this represents a limitation of our work, it is probably an inevitable choice given our initial goals, i.e., to involve many users ranging from the “newbie” to the “elite” hacker and compare them on the same set of challenges. We hope that future studies will either confirm (or disprove) our results with larger and more difficult binaries to reverse. For example, we can hypothesize a more frequent use of the birdseye overview (described in Section 5.5.3), which in our experiments was used only by a small percentage of experts. Another aspect that is largely related to the size of the binaries is the number of functions, and therefore we expect a more pronounced impact of the different strategies described in Section 5.5.1 on larger programs. For instance, an initial horizontal investigation can be beneficial when analyzing larger codebases.

5.8 Conclusions

Drawing inspiration from the first set of interviews conducted by Votipka in 2020 [VRM+20], the objective of our study was to lay the second brick towards a solid understanding of the RE process from an assembly code comprehension perspective.

A deep understanding of the topic can help us from different points of view and has a few interesting implications that should be taken into account. With our work, we hope to provide a valuable input for future research in a field that, so far, was poorly explored.

In the spirit of open science, we release [3] the source code of our web RE framework together with the challenges and the test scripts, to allow the community to continue further studies in this direction. Lastly, our measurements are summarized in Table 5.8.

[3]https://github.com/elManto/REmind

Table 5.8: Individual Experts Features

User	Solution Time	Function Exploration (Test1 ; Test2)	Transitions	Tfirst=Ttot	Tfirst=Tmax	Skipped BB
Exp.1	169	Forward,Hybrid;Forward,DFS	2997	16.7%	14.8%	18
Exp.2	137	Forward;Hybrid;Forward,BFS	2551	15.4%	8.3%	19
Exp.3	120	Forward,BFS;Forward,DFS	1662	29.6%	12.9%	24
Exp.4	120	Backward,Hybrid;Backward,BFS	978	29.8%	16.7%	30
Exp.5	163	Forward,Hybrid;Forward,BFS	1654	35.6%	17.2%	4
Exp.6	137	Forward,Hybrid;Forward,Hybrid	2359	18.7%	14.1%	15
Exp.7	134	Forward,Hybrid;Forward,DFS	2845	22.5%	8.3%	21
Exp.8	119	Forward,Hybrid;Forward,Hybrid	1750	29.6%	7.7%	30
Exp.9	156	Forward,Hybrid;Forward,Hybrid	2070	13.5%	21.2%	4
Exp.10	162	Backward,Hybrid;Backward,Hybrid	2141	25.1%	9.6%	24
Exp.11	37	Forward,Hybrid;Forward,Hybrid	450	46.4%	14.8%	31
Exp.12	34	Backward,Hybrid;Backward,Hybrid	727	35.4%	16.7%	25
Exp.13	118	Forward,BFS;Backward;Hybrid	1546	9.6%	18.7%	2
Exp.14	119	Forward,BFS;Forward,BFS	1842	11.6%	14.8%	4
Exp.15	96	Forward,BFS;Forward,BFS	1564	19.3%	17.4%	23
Exp.16	154	Forward,Hybrid;Forward,BFS	2547	10.9%	14.8%	1
Exp.17	80	Forward,BFS;Forward,DFS	1321	23.8%	16.7%	13
Exp.18	48	Forward,Hybrid;Forward,BFS	761	34.8%	10.9%	41
Exp.19	81	Forward,BFS;Backward,BFS	746	40.0%	13.8%	3
Exp.20	48	Forward,Hybrid;Forward,DFS	818	22.5%	21.2%	34
Exp.21	38	Forward,BFS;Forward,BFS	483	47.7%	6.4%	27
Exp.22	43	Backward,DFS;Backward,Hybrid	627	41.2%	20.0%	29
Exp.23	44	Forward,Hybrid;Forward,Hybrid	673	39.3%	16.7%	32
Exp.24	27	Backward,Hybrid;Backward,DFS	462	46.4%	20.6%	30
Exp.25	29	Forward,Hybrid;Forward,Hybrid	634	36.1%	23.8%	45
Exp.26	45	Forward,Hybrid;Forward,Hybrid	789	35.4%	18.0%	27
Exp.27	64	Forward,Hybrid;Forward,Hybrid	835	45.1%	11.6%	21
Exp.28	70	Forward,Hybrid;Forward,BFS	1478	31.6%	10.9%	24
Exp.29	32	Forward,Hybrid;Forward,Hybrid	820	38.7%	14.1%	23
Exp.30	89	Backward,DFS;Backward,Hybrid	1415	14.1%	15.4%	8
Exp.31	171	Forward,Hybrid;Forward,BFS	2115	10.9%	10.9%	26
Exp.32	56	Forward,BFS;Forward,BFS	517	45.8%	14.8%	44
Exp.33	106	Forward,Hybrid;Forward,Hybrid	988	38.0%	17.4%	24
Nov.1	266	Forward,Hybrid;Backward,Hybrid	3330	5.8%	10.3%	2
Nov.2	329	Forward,Hybrid;Forward,BFS	5114	7.7%	6.4%	7
Nov.3	304	Forward,BFS;Forward,BFS	5025	14.1%	8.3%	13
Nov.4	208	Forward,Hybrid;Forward,BFS	3793	12.9%	9.0%	9
Nov.5	260	Forward,Hybrid;Forward,Hybrid	3560	20.0%	5.1%	19
Nov.6	257	Forward,Hybrid;Forward,DFS	4136	8.3%	8.3%	4
Nov.7	264	Forward,Hybrid;Backward,Hybrid	3433	5.1%	9.0%	1
Nov.8	331	Forward,Hybrid;Forward,Hybrid	3805	3.2%	10.9%	0
Nov.9	303	Forward,Hybrid;Forward,Hybrid	3892	16.1%	9.6%	20
Nov.10	415	Forward,Hybrid;Forward,BFS	7602	12.2%	3.8%	10
Nov.11	371	Forward,BFS;Forward,BFS	4796	9.6%	10.3%	8
Nov.12	381	Forward,Hybrid;Backward,Hybrid	4514	16.1%	9.1%	17
Nov.13	258	Forward,Hybrid;Forward,BFS	2374	1.2%	10.3%	0
Nov.14	458	Sequential,Hybrid;Sequential,Hybrid	6955	4.5%	6.4%	1
Nov.15	251	Forward,Hybrid;Backward,BFS	3067	24.5%	14.8%	22
Nov.16	409	Forward,BFS;Forward,BFS	5656	4.5%	9.6%	2
Nov.17	481	Forward,BFS;Sequential,BFS	6270	8.3%	4.5%	9
Nov.18	560	Backward,Hybrid;Sequential,Hybrid	6976	5.1%	7.1%	1
Nov.19	194	Forward,Hybrid;Forward,Hybrid	2791	7.7%	13.5%	1
Nov.20	351	Forward,Hybrid;Forward,Hybrid	3685	4.5%	10.3%	1
Nov.21	301	Forward,Hybrid;Forward,Hybrid	5020	10.3%	10.9%	3
Nov.22	228	Backward,BFS;Backward,Hybrid	2841	16.1%	12.2%	19
Nov.23	300	Forward,BFS;Forward,Hybrid	3091	7.7%	12.3%	6
Nov.24	195	Forward,Hybrid;Forward,Hybrid	2592	6.4%	14.1%	0
Nov.25	261	Forward,Hybrid;Forward,BFS	2510	11.6%	13.5%	3
Nov.26	240	Forward,BFS;Forward,BFS	3050	15.4%	9.6%	11
Nov.27	740	Sequential,Hybrid;Sequential,Hybrid	7879	6.4%	13.1%	8
Nov.28	543	Forward,Hybrid;Sequential,Hybrid	6641	2.5%	7.7%	0
Nov.29	941	Sequential,Hybrid;Sequential,Hybrid	8150	0.0%	1.9%	0
Nov.30	320	Forward,BFS;Forward,BFS	2912	12.5%	7.1%	16
Nov.31	316	Forward,Hybrid;Forward,BFS	2886	18.7%	10.9%	13
Nov.32	234	Forward,Hybrid;Forward,BFS	4048	7.7%	6.4%	3
Nov.33	181	Forward,BFS;Forward,BFS	3445	12.2%	8.3%	10
Nov.34	207	Forward,Hybrid;Forward,Hybrid	2202	29.0%	9.6%	26
Nov.35	513	Forward,BFS;Sequential;Forward,BFS	6233	0.0%	4.5%	0
Nov.36	199	Forward,Hybrid;Forward,Hybrid	3408	18.7%	6.4%	17
Nov.37	178	Forward,BFS;Forward,Hybrid	1618	5.8%	19.3%	3
Nov.38	441	Forward,Hybrid;Sequential,Hybrid	5946	6.4%	8.3%	1
Nov.39	253	Forward,Hybrid;Forward,Hybrid	3486	21.9	10.9%	7

Chapter 6

Future Work and Conclusion

6.1 Future Work

One of the main takeaways of this thesis is that while complex binary analysis tasks can already be fully automated, some form of human effort is still needed at different points of the pipeline. This raises a number of challenges that fall into two main lines of research, one focusing on the human side and the other attempting to improve the current automation of binary analysis.

6.1.1 Human studies and Binary Analysis

The first area that is directly influenced by this thesis is teaching. More specifically, we believe that by leveraging some of the aspects presented in the last chapter, other researchers could propose novel methods to improve the learning phase, especially w.r.t. reverse engineering. Several features we identified correlate with experience, but this does not mean that we cannot improve them by performing specific exercises. As of now, the learning phase is mainly driven by the solution of binary challenges of increasing complexity [CN15]. A possible implication of our findings could be to design binary analysis exercises more focused on a few basic blocks to stimulate a student to match and memorize specific patterns. Similarly, this aspect can influence the formation of experts that work in other domains of binary analysis. For instance, in malware analysis, we could propose specific training exercises to refine the skill of distinguishing between malicious and benign files.

A different line of work could instead focus on improving the knowledge about the RE mental process as performed by human analysts. Future research, therefore, might study the many aspects that remained uninvestigated, thus offering a broader range of findings and insights on the way experts binary analysts work. For instance, an interesting follow-up of our work would be to design a set of experiments specifically for expert participants, thus including more complex tasks and challenges.

Besides that, another branch of research could focus on other aspects of a more specific domain, such as malware analysis or vulnerability discovery. Indeed, we believe that each expert develops a unique set of skills linked to the specific task that she faces every day. Also, the methodology will play a fundamental role, preferring remotely accessible solutions for studies over a large group or eye-tracking devices for smaller groups.

6.1.2 Machines and Binary Analysis

This thesis demonstrates that machines still lack the needed skills to “comprehend” the code, thus requiring the help of humans to perform this task.

This suggests that augmenting modern tools and approaches with this single capability could be a promising road for the future of binary analysis.

Indeed, if teaching binary analysis to humans is essential to form new experts in the domain, training computers to mimic human behavior would be fundamental to scale over the large amount of software (both benign and malicious) released every day. We believe that studying the techniques used by humans is the first step to discovering new ways to train machine learning models to perform similar tasks. Psychologists have learned that many activities are inherently linked to the ability to recognize previously seen patterns [WUF97, FM98, LS91] and that the experts are those who learned a significant number of patterns over several years of experience. Since learning to recognize patterns is what ML algorithms can do well, in this thesis, we also studied which aspects human experts focus their attention on to provide the building block for further studies on such topics. Extending the concept, we could even introduce semantic awareness in the classifier. For instance, many experts in our experiments could easily recognize non-standard implementations of list operations or discard branches/functions by just reading a subset of their basic blocks. This suggests that ML classifiers could be trained to mimic this behavior and to automate the pattern recognition phase both for useful and useless portions of code.

Another possibility instead is to focus on binary analysis tools that are more robust w.r.t. binary code. For instance, as we show in Chapter 4, the main weakness of current static analyzers is that they are designed to analyze only well-formed source code. However, we believe that with the constant enhancement of modern decompilers, their adoption could represent an opportunity to automate other security-related approaches, different from static software testing, once proper tools are implemented to deal with pseudocode. Similarly, in the future, decompilers should be projected to allow different analysis paradigms and thus not to assume a-priori that their output is only designed to increase readability for humans.

On the other hand, we may never be able to replace humans with automated machines completely. Thus, another research line will consist of usability studies for binary analysis interfaces. So far, this area has been mostly ignored, but even if our studies do not explicitly speak about usability concepts, we believe this last point deserves its own discussion. For instance, the last chapter 5, shows how most of the participants choose the branch depending on the position on the screen, demonstrating how the visual component affects the RE session.

6.2 Conclusion

In this thesis, we proposed a snapshot of the current binary analysis approaches and demonstrated how standard techniques are still heavily based on human intervention. For each of the proposed methodologies, we emphasized what the human activity is and how this is fundamental to help the automated components to achieve a certain goal. Our categorization showed that analysts could have different positions in the analysis pipeline, and this requires different roles and skills depending on the scenario. We believe that future research should spend more effort on this human-centric view because this could deliver essential benefits to the entire field of binary analysis.

To conclude, we hope that our work can shed light on this poorly investigated aspect that surrounds the binary analysis world and that future work will consider our findings to increase the understanding of the binary analysis ecosystem and the implementation of novel, more automated methodologies.

Appendices

Appendix A

REmind

A.1 Text of the invitation email

Experiment purpose

A study about how humans coming from different backgrounds and expertise levels (from the 'noob' to 'expert') perform the process of Reverse Engineering and which are the main differences between these categories.

Before starting

The test is completely anonymous, the registration is mandatory but it is quick (just a self-evaluation question). The system will give you a token which is needed for the login, so please preserve it until the end of the test.

The test

For the test, you can find our web-UI at this link (`https://reverse.s3.eurecom.fr`): it supports some of the main features for RE (commenting code, rename, Xref, ...). After accessing it, the first page comes with a further description of the experiment and of the interface (we invite you to read for the details) and with a list of 3 challenges that you have to solve with our web-UI. The first challenge ('Warmup') is just a warmup one so it is optional and we created it just to make the user become more familiar with the tool. The second and the third challs (namely 'Test 1' and 'Test 2') represent the core part of the experiment. Clicking on one of the two tests starts the RE interface. From now on, your job consists of understanding what the binary does and then submitting a solution in the proper form. You can solve the 2 tests in separate moments and you can stop a RE session and then re-start it (even if we think the best thing is that you stop the RE session after submitting a solution).

Submitting a solution

Note that for the two tests (Test 1 and Test 2), the solution is not required in a specific format (like the flags in a CTF), but it is supposed to be a short description in your own words (just 1 or 2 lines) about the needed steps that make the binary to print the string 'Congratulations' or 'Success!'. Alternatively, also a command line that triggers the correct path in the binary is fine.

Notes

- The interface is not supposed to be a new competitive product, but it is just a tool for the data collection. This does NOT aim to be a "realistic scenario", but a "scenario for which we capture some interesting data". It's an experiment! Please bear with it :-)
- The experience could result a bit painful because basic blocks are blurred when the 'onmouseover' event is captured on another BB. Al-

though we fully understand this makes the RE process slower, this is needed for some aspects we are trying to collect. So, yes, this is NOT your IDA experience you are looking for... it's an experiment! Please bear with it #2 :-)

- For the tests ('Test 1', and 'Test 2'), we disabled the 'Strings' view. Also in this case, the reason is linked with our models and the data we need to collect. So do not worry if you cannot access the 'Strings' view, there is no bug, it is just a design choice.

- In general, we are interested in static analysis, not dynamic one. This explains why we did not add a debugger to the tool. If you are used to reverse with a debugger, that's good! But for this experiment we are interested to know how you would approach a purely static analysis task!

- There is no ranking or prize, this is just an experiment: so please do not cheat.

Thanks a lot for your time/help!

References

[AA14] Andronicus A Akinyelu and Aderemi O Adewumi. Classification of phishing email using random forest machine learning technique. *Journal of Applied Mathematics*, 2014, 2014.

[ACC+17] A. Arusoaie, S. C., V. Craciun, D. Gavrilut, and D. Lucanu. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *IEEE SYNASC*, 2017.

[AM14] H.H. AlBreiki and Q.H. Mahmoud. Evaluation of static analysis tools for software security. In *IIT*, 2014.

[aptay] Apt and cybercriminal campaign collection. https://github.com/CyberMonitor/APT_CyberCriminal_Campagin_Collections, Accessed April 20, 2022.

[AS96] Vairam Arunachalam and William Sasso. Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering. *Journal of Systems and Software*, 1996.

[ASPE13] Rohit Arora, Anishka Singh, Himanshu Pareek, and Usha Rani Edara. A heuristics-based static analysis approach for detecting packed pe binaries. *International Journal of Security and Its Applications*, 7(5):257–268, 2013.

[avaay] Avast retargetable decompiler ida plugin. https://blog.fpmurphy.com/2017/12/avast-retargetable-decompiler-ida-plugin.html, Accessed April 20, 2022.

[AvML98] A Anneliese von Mayrhauser and Steve Lang. Program comprehension and enhancement of software. In *In Proceedings IFIP World Computing Congress-Information Technology and Knowledge Engineering*, 1998.

[Ban] Liam J Bannon. From human factors to human actors: The role of psychology and human-computer interaction studies in system design. In *Readings in human–computer interaction*. Elsevier.

[BB10] Jan Berdajs and Zoran Bosnić. Extending applications using an advanced approach to dll injection and api hooking. *Software: Practice and Experience*, 40(7):567–584, 2010.

[BCCO16] C. Barria, D. Cordero, C. Cubillos, and R. Osses. Obfuscation procedure based in dead code insertion into crypter. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*, pages 23–29, May 2016.

[BK] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*.

[bloaya] C and c++ source code analysis tools. https://www.codeanalysistools.com/?cplusplus, Accessed April 20, 2022.

[bloayb] What are the best sast tools? https://cybersecuritykings.com/2020/02/16/11-tips-on-sast-tool-selection/, Accessed April 20, 2022.

[BLSW13] D. Brumley, J. Lee, E.J. Schwartz, and M. Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In {*USENIX*}, 2013.

[BM08] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, Aug 2008.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 1983.

[Bry12] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. PhD thesis, Air Force Institute Of Technology, 2012.

[BT04] Roman Bednarik and Markku Tukiainen. Visual attention tracking during program debugging. In *Proceedings of the third Nordic conference on Human-computer interaction*, 2004.

[BT06] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[BZRL12] M. Baig, P. Zavarsky, R. Ruhl, and D. Lindskog. The study of evasion of packed pe from static detection. In *World Congress on Internet Security (WorldCIS-2012)*, pages 99–104, June 2012.

[capay] Capstone. http://www.capstone-engine.org/, Accessed April 20, 2022.

[CC19] Taylor Claire and Christian Collberg. Getting revenge: A system for analyzing reverse engineering behavior. 2019.

[CFBS15] Jill Cao, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. Idea garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 2015.

[CG] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience.*

[CG07] Edward Cutrell and Zhiwei Guan. What are you looking for? an eye-tracking study of information usage in web search. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007.

[cheay] Checkmarx. https://www.checkmarx.com/, Accessed April 20, 2022.

[CK11] G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of c code vulnerabilities. In *IEEE COMPSAC*, 2011.

[CkKOcR08] Yang-Seo Choi, Ik kyun Kim, Jin-Tae Oh, and Jae cheol Ryou. Pe file header analysis-based packed pe file detection technique (phad). *International Symposium on Computer Science and its Applications*, pages 28–31, 2008.

[CM] B. Chess and G. McGraw. Static analysis for security. *2004 IEEE S&P.*

[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. ACM, 2018.

[CN15] Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In {*USENIX*} *(3GSE 15)*, 2015.

[codaya] Code-ql. https://securitylab.github.com/tools/codeql, Accessed April 20, 2022.

[codayb] Code-ql queries examples. https://help.semmle.com/QL/learn-ql/cpp/ql-for-cpp.html, Accessed April 20, 2022.

[covay] Coverity. https://scan.coverity.com/, Accessed April 20, 2022.

[cppay] Cppcheck. http://cppcheck.sourceforge.net/, Accessed April 20, 2022.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[cweay] Cwe checker. https://github.com/fkie-cat/cwe-checker, Accessed April 20, 2022.

[DAL+17] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[DARay] DARPA. Darpa celebrates cyber grand challenge winners. https://www.darpa.mil/news-events/2016-08-05a, Accessed April 20, 2022.

[DBXP20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[DC09] E.N. Dolgova and A.V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 2009.

[defay] Defcon ctf final scores. https://www.defcon.org/html/defcon-24/dc-24-ctf.html, Accessed April 20, 2022.

[detay] Detect-it-easy signatures. https://github.com/horsicq/Detect-It-Easy, Accessed April 20, 2022.

[DGHH$^+$15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2015.

[DN12] Dhruwajita Devi and Sukumar Nandi. Pe file features in detection of packed executables. *International Journal of Computer Theory and Engineering*, 4(3):476, 2012.

[DPY18] Y. David, N. Partush, and E. Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 2018.

[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.

[DR14] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon*, 2014.

[EN08] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 2008.

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):1–42, 2008.

[Fat04] Holy Father. Hooking windows api-technics of hooking api functions on windows. *CodeBreakers J*, 1(2), 2004.

[FBH18] A. Fatima, S. Bibi, and R. Hanif. Comparative study on static code analysis tools for c/c++. In *IEEE IBCAST*, 2018.

[FCL+19] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3708–3719, 2019.

[flaay] flawfinder. https://github.com/david-a-wheeler/flawfinder, Accessed April 20, 2022.

[FM98] Lev Finkelstein and Shaul Markovitch. Learning to play chess selectively by acquiring move patterns. *ICGA Journal*, 1998.

[FMB+19] J. Feist, L. Mounier, S. Bardin, R. David, and M. Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *SSPREW*, 2019.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[For07] Behrouz A Forouzan. *Cryptography & network security*. McGraw-Hill, Inc., 2007.

[foray] Fortify sca. https://www.microfocus.com/en-us/products/static-code-analysis-sast, Accessed April 20, 2022.

[fraay] framac. https://frama-c.com/, Accessed April 20, 2022.

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

[FWS93] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, 1993.

[gdb] Gdb: the gnu project debugger. https://www.gnu.org/software/gdb/current/.

[GDFFA20] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta. A comb for decompiled c code. In *ACM AsiaCCS*, 2020.

[ghiay] Ghidra. https://ghidra-sre.org/, Accessed April 20, 2022.

[GJC+03] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and David V. Buffer overrun detection using linear programming and static analysis. In *ACM CCS*, 2003.

[GO86] Leo Gugerty and Gary Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*, 1986.

[greay] The new malicious software grayenergy. https://cert.gov.ua/news/45, Accessed April 20, 2022. Accessed April 20, 2022, Ukrainian language.

[HDWY06] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[hexay] Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/, Accessed April 20, 2022.

[HGK03] Allyson L Holbrook, Melanie C Green, and Jon A Krosnick. Telephone versus face-to-face interviewing of national probability samples with long questionnaires: Comparisons of respondent satisficing and social desirability response bias. *Public opinion quarterly*, 2003.

[HL09] Seung-Won Han and Sang-Jin Lee. Packed pe file detection for malware forensics. *The KIPS Transactions: PartC*, 16(5):555–562, 2009.

[Hoc14] J-M Hoc. *Psychology of programming*. Academic Press, 2014.

[HP07] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *ACM SIGPLAN-SIGSOFT PASTE*, 2007.

[HPY+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 248–258, 2014.

[HSP05] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN-SIGSOFT PASTE*, 2005.

[idaay] Ida pro. https://www.hex-rays.com/products/ida/, Accessed April 20, 2022.

[ikoay] Ikos. https://github.com/NASA-SW-VnV/ikos, Accessed April 20, 2022.

[infay] Infer. https://fbinfer.com/, Accessed April 20, 2022.

[JBM03] Anthony R Jansen, Alan F Blackwell, and Kim Marriott. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, 2003.

[JCN+12] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2012.

[JLS+18] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *IEEE/ACM ICPC*, 2018.

[joeaya] Joern. https://joern.io/, Accessed April 20, 2022.

[joeayb] Joern queries examples. https://github.com/ShiftLeftSecurity/joern/tree/master/joern-cli/src/main/resources/scripts/c, Accessed April 20, 2022.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 3–9. IEEE, 2015.

[JST13] Joby James, L Sandhya, and Ciza Thomas. Detection of phishing urls using machine learning techniques. In *ICCC*. IEEE, 2013.

[Kap96] Victor Kaptelinin. Activity theory: Implications for human-computer interaction. *Context and consciousness: Activity theory and human-computer interaction*, 1996.

[KKK$^+$20] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *ACSAC*, 2020.

[KLHC10] Y. Kim, J. Lee, H. Han, and K. Choe. Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 2010.

[KMCA06] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 2006.

[KMZ17] J. Křoustek, P. Matula, and P. Zemek. Retdec: An open-source machine-code decompiler, 2017.

[KOGY19] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *In Proc. ACM Workshop Recurring Malcode (WORM*, pages 46–53. ACM, 2007.

[Kra05] K.J. Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Technical report, HARVARD UNIV CAMBRIDGE MA, 2005.

[KRS18] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *IEEE SANER*, 2018.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[LAB11] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[Laf04] Eric Lafortune. Proguard. https://sourceforge.net/projects/proguard/, 2004. Accessed April 20, 2022.

[LAHR10] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code.* Wiley Publishing, 2010.

[LC10] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE international conference on information theory and information security*, pages 521–524. IEEE, 2010.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[Let87] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 1987.

[LGHM07] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[LH07] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[LK09] Tímea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, 30:3–19, 08 2009.

[LKJ+16] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: effects of explicit guidance. In *CHI Conference on Human Factors in Computing Systems*, 2016.

[LLZW17] H. Liang, S. Liu, Y. Zhang, and M. Wang. Improving the precision of static analysis: Symbolic execution based on gcc abstract syntax tree. In *SNPD*, 2017.

[LPLS87] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 1987.

[LRK$^+$19] Charles Lim, Kalamullah Ramli, Yohanes Syailendra Kotualubun, et al. Mal-flux: Rendering hidden code of packed binary executable. *Digital Investigation*, 28:83–95, 2019.

[LS91] Robert Levinson and Richard Snyder. Adaptive pattern-oriented chess. In *Machine Learning Proceedings 1991*. Elsevier, 1991.

[LSCL12] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012.

[LW20] Z. Liu and S. Wang. How far we have come: testing decompilation correctness of c decompilers. In *SIGSOFT ISSTA*, 2020.

[LYS$^+$19] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. Dire: A neural approach to decompiled identifier naming. In *IEEE/ACM ASE*, 2019.

[manay] Manalyze. https://github.com/JusticeRage/Manalyze, Accessed April 20, 2022.

[MBC17] Philip Menard, Gregory J Bott, and Robert E Crossler. User motivations in protecting information security: Protection motivation theory versus self-determination theory. *Journal of Management Information Systems*, 2017.

[MC06] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, Apr 2006.

[McL12] R. K McLean. Comparing static security analysis tools using open source software. In *IEEE SERE*, 2012.

[MDC17] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61, 2017.

[MHH$^+$19] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[MHK08] Daisuke Miyamoto, Hiroaki Hazeyama, and Youki Kadobayashi. An evaluation of machine learning-based methods for detection of phishing sites. In *International Conference on Neural Information Processing*. Springer, 2008.

[Mic] Microsoft. Microsoft store top free apps. https://www.microsoft.com/en-us/store/top-free/apps/pc. Accessed April 20, 2022.

[MJZ$^+$15] S. Ma, M. Jiao, S. Zhang, W. Zhao, and D.W. Wang. Practical null pointer dereference detection via value-dependence analysis. In *IEEE ISSREW*, 2015.

[MM18] R. Mahmood and Q.H. Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040*, 2018.

[MP10] Maik Morgenstern and Hendrik Pilz. Useful and useless statistics about viruses and anti-virus programs. In *Proceedings of the CARO Workshop*, 2010.

[NLC16] M. Noonan, A. Loginov, and D. Cok. Polymorphic type inference for machine code. In *ACM SIGPLAN PLDI*, 2016.

[NNTH$^+$21] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[OEV$^+$] Mete Ozay, Inaki Esnaola, Fatos Tunay Yarman Vural, Sanjeev R Kulkarni, and H Vincent Poor. Machine learning methods for attack detection in the smart grid. *IEEE transactions on neural networks and learning systems*.

[OMNER19] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

[OSM11] P. OKane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security Privacy*, 9(5):41–47, Sep. 2011.

[PAP$^+$21] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *ICSE*. IEEE, 2021.

[pefay] pefile. https://github.com/erocarrera/pefile, Accessed April 20, 2022.

[peiay] Peid. https://www.aldeid.com/wiki/PEiD, Accessed April 20, 2022.

[Pen87] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 1987.

[PF20] S. Poeplau and A. Francillon. Symbolic execution with symcc: Don't interpret, compile! In {*USENIX*}, 2020.

[PGG$^+$15] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *IEEE S&P*, 2015.

[PHG$^+$04] Bing Pan, Helene A Hembrooke, Geri K Gay, Laura A Granka, Matthew K Feusner, and Jill K Newman. The determinants of web page viewing behavior: an eye-tracking study. In *Proceedings of the 2004 symposium on Eye tracking research & applications*, 2004.

[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 *CyberSA*. IEEE, 2015.

[PLL08a] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 301–310, Dec 2008.

[PLL08b] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern recognition letters*, 29(14):1941–1946, 2008.

[PSDV06] D. Pozza, R. Sisto, L. Durante, and A. Valenzano. Comparing lexical analysis tools for buffer overflow detection in network software. In *COMSWARE*, 2006.

[ratay] Rats. https://code.google.com/archive/p/rough-auditing-tool-for-security/, Accessed April 20, 2022.

[RBP17] B. Rahbarinia, M. Balduzzi, and R. Perdisci. Exploring the long tail of (malicious) software downloads. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 391–402, June 2017.

[RHD$^+$06] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, Dec 2006.

[RJP08] Keith Rayner, Barbara J. Juhasz, and Alexander Pollatsek. *Eye Movements During Reading*. John Wiley and Sons, Ltd, 2008.

[RM13] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[RSCC04] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, pages 22–es, 2004.

[RTKM12] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *34th ICSE*. IEEE, 2012.

[RV15] Jithu Raphel and P. Vinod. Information theoretic method for classification of packed and encoded files. In *Proceedings of the 8th International Conference on Security of Information* *and Networks*, SIN '15, pages 296–303, New York, NY, USA, 2015. ACM.

[sasay] Awesome static analysis. https://github.com/analysis-tools-dev/static-analysis, Accessed April 20, 2022.

[scaay] Scan-build. https://clang-analyzer.llvm.org/, Accessed April 20, 2022.

[SEHM13] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 2013.

[SFM12] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2012.

[SH12] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[SKBM06] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 2006.

[SMM15] S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In *IEEE ISSREW*, 2015.

[SOA18] Rami Sihwail, Khairuddin Omar, and Khairul Akram Zainol Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2):1662, 2018.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[SRN$^+$18] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov. Evolving exact decompilation. In *BAR*, 2018.

[SRX14] M. Saleh, E. P. Ratazzi, and S. Xu. Instructions-based detection of sophisticated obfuscation and packing. In *2014 IEEE Military Communications Conference*, pages 1–6, Oct 2014.

[Ste05] Adrian E Stepan. Defeating polymorphism: beyond emulation. In *Proceedings of the Virus Bulletin International Conference*, 2005.

[STF09] Muhammad Zubair Shafiq, S. Momina Tabish, and Muddassar Farooq. Pe-probe : Leveraging packer detection and structural information to detect malicious portable executables. 2009.

[STMF09] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SUPS$^+$11] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 23–30. ACM, 2011.

[SWD$^+$17] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM SIGSAC CCS*, 2017.

[SYS$^+$08] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 481–500, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[TFSL14] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2014.

[THLL09] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 2009.

[TY07] Roger Tourangeau and Ting Yan. Sensitive questions in surveys. *Psychological bulletin*, 2007.

[TZ09] Scott Treadwell and Mian Zhou. A heuristic approach for detection of obfuscated malware. pages 291–299, 01 2009.

[UAB19] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81, 2019.

[UNMM06] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[UPBSB15] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 659–673. IEEE, 2015.

[UPSGF$^+$14] Xabier Ugarte-Pedrero, Igor Santos, Iván García-Ferreira, Sergio Huerta, Borja Sanz, and Pablo G Bringas. On the adoption of anomaly detection for packed executable filtering. *Computers & Security*, 43:126–144, 2014.

[UPSS$^+$12] Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. Countering entropy measure attacks on packed software detection. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 164–168. IEEE, 2012.

[VBKM] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *2000 ACSAC.*

[veray] Veracode. https://www.veracode.com/products/binary-static-analysis-sast, Accessed April 20, 2022.

[viray] Virus total. https://www.virustotal.com/, Accessed April 20, 2022.

[VRM$^+$20] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX}*, 2020.

[VSR$^+$18] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *IEEE SP*. IEEE, 2018.

[WFBA00] D. A. Wagner, J. S Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[WFS93] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 1993.

[WUF97] Andrew J Waters, Geoffrey Underwood, and John M Findlay. Studying expertise in music reading: Use of a pattern-matching paradigm. *Perception & psychophysics*, 59(4), 1997.

[XGM08] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA*, 2008.

[yar] Yara rules. https://github.com/Yara-Rules.

[YDGPS16] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *IEEE S&P*, 2016.

[YEGPS15] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

[YGAR] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE S&P*.

[YLR12] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC*, 2012.

[YSCX17] H. Yan, Y. Sui, S. Chen, and J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *ACSAC*, 2017.

[YSCX18] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE*, 2018.

[Yur13] Dennis Yurichev. Reverse engineering for beginners, 2013.

[YZA08] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. *IEEE Security Privacy*, 6(5):65–69, Sep. 2008.

[YZH14] J. Ye, C. Zhang, and X. Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *ACM CCS*, 2014.

[ZEKAS17] Marwane Zekri, Said El Kafhali, Noureddine Aboutabit, and Youssef Saadi. Ddos attack detection using machine learning techniques in cloud computing environments. In *3rd CloudTech*. IEEE, 2017.