ST. PÖLTEN UNIVERSITY
OF APPLIED SCIENCES

# Implementation and evaluation of a continuous Just-In-Time obfuscator

Master thesis

For attainment of the academic degree of

Master of Science in Engineering (MSc)

submitted by

JORDI ZAYUELAS I MUÑOZ
cr211529

in the
University Course Cyber Security and Resilience at St. Pölten University of Applied Sciences

Supervision
Advisor: Dipl.-Ing. Patrick Kochberger, BSc
Assistance: -

St. Pölten, September 20, 2023

(Signature author) (Signature advisor)

# Declaration

I hereby affirm that

- I have written this thesis independently, that I have not used any sources or aids other than those indicated, and that I have not made use of any unauthorised assistance.
- I have not previously submitted this thesis topic to an assessor, either in Austria or abroad, for evaluation or as an examination paper in any form.
- this thesis corresponds to the thesis assessed by the assessor.

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

*Date* *Signature*

# Abstract

In the realm of computer security, code obfuscation is a commonly used technique employed by various entities, each driven by distinct motives. Corporations utilize code obfuscation as a means to safeguard their software code and proprietary algorithms from potential reverse engineering attempts. Conversely, malevolent actors exploit this technique to conceal their malicious software from detection by antivirus programs or to impede analysis by security teams upon discovering the malware. An effective obfuscator must substantially increase the complexity of reverse-engineering a program while preserving its original semantics, all while maintaining a reasonable performance overhead.

Among the existing methods used for code obfuscation, analysing self-modifying code can be challenging, since it is difficult to debug and understand, rendering it an ideal candidate for obfuscating code to shield it against analysis. To validate this assertion, an obfuscation technique utilizing self-modifying code to obscure the real instructions of the program code was conceptualized, implemented within a custom packer, and subjected to rigorous evaluation. The obfuscation method operates by employing an external function that dynamically reconstructs basic code blocks of the program as they are required, and conceals them once their execution is complete. Consequently, only a small portion of the authentic code is in memory at any given time.

The results of the evaluation attest to the resilience and potency of the proposed obfuscation method. Despite introducing a notable execution time overhead, the technique exhibits good results and poses as a viable means to safeguard critical functions of programs, particularly in scenarios where optimal efficiency is not of paramount concern. Further research and refinement of this technique hold substantial potential for enhancing the security landscape in the ever-evolving domain of computer security.

# Contents

**1 Introduction** . . . . . **1**

1.1 Research Questions . . . . . 2

1.2 Thesis Outline . . . . . 2

**2 Prerequisites** . . . . . **3**

2.1 Executable files . . . . . 3

2.2 ELF File format . . . . . 3

2.2.1 Sections . . . . . 4

2.2.2 Segments . . . . . 4

2.2.3 Position Independent Executables . . . . . 5

2.3 Assembly . . . . . 6

2.4 Reverse Engineering . . . . . 6

2.4.1 Basic structures . . . . . 6

2.4.2 Static Analysis . . . . . 7

2.4.3 Dynamic Analysis . . . . . 7

2.4.4 Automated Analysis . . . . . 8

2.4.5 Manual Analysis . . . . . 8

2.4.6 Symbolic execution . . . . . 8

2.4.7 Reverse engineering tools . . . . . 8

2.5 Self-Modifying Code . . . . . 9

2.6 Code Obfuscation . . . . . 9

2.6.1 Obfuscation techniques . . . . . 10

2.6.2 Evaluation metrics . . . . . 10

**3 Related Work** . . . . . **13**

3.1 Reverse engineering . . . . . 13

3.2 Obfuscation methods . . . . . 15
3.3 Discussion . . . . . 17

**4 Concept** . . . . . **19**
4.1 Challenges . . . . . 21
4.2 Benefits . . . . . 23
4.3 Weaknesses . . . . . 23
4.4 Countermeasures . . . . . 24

**5 Implementation** . . . . . **27**
5.1 Technical Requirements . . . . . 28
5.2 Overview . . . . . 28
5.3 Initial analysis . . . . . 30
5.4 Header modification . . . . . 30
5.5 Second analysis . . . . . 30
5.6 Code block transformation . . . . . 31
5.7 Store data . . . . . 32
5.8 Add transformation function . . . . . 33
5.9 Additional functionalities . . . . . 33

**6 Evaluation** . . . . . **37**
6.1 Data Collection . . . . . 37
6.2 Size . . . . . 40
6.3 Execution Time . . . . . 40
6.3.1 Static Analysis . . . . . 41
6.3.2 Dynamic Analysis . . . . . 41
6.4 Results . . . . . 43
6.4.1 Potency . . . . . 43
6.4.2 Resilience . . . . . 44
6.4.3 Costs . . . . . 45
6.4.4 Stealth . . . . . 45
6.5 Comparison with previous similar studies . . . . . 46

**7 Conclusion** . . . . . **49**

7.1 Future work . . . 50

**List of Figures** . . . **51**

**List of Tables** . . . **52**

**Acronyms** . . . **55**

**Bibliography** . . . **57**

# 1 Introduction

The importance of safeguarding sensitive information and intellectual property within source code cannot be overstated. In today's digital age, where technology serves as the backbone of countless industries, the significance of secure software development practices has grown exponentially. Companies and institutions invest substantial resources in developing software, which often forms a crucial part of their internal operations or is presented as a marketable product. These programs and underlying algorithms represent a significant value, making them enticing targets for malicious actors aiming to exploit vulnerabilities or reverse-engineer proprietary algorithms. Consequently, ensuring the robust security of these digital assets has become a paramount concern for software developers and organizations alike.

When it comes to the transmission of information, Cryptography stands as a fundamental tool for protecting data from being intercepted and recovered by unwanted parties. It effectively conceals information during communication, making it exceedingly difficult to decipher without the appropriate decryption keys. However, this cryptographic approach, while highly effective for secure communication, cannot be directly applied to protect publicly released software from analysis. When software is distributed for execution, the processor needs access to the code, essentially requiring exposure of the encryption keys.

To safeguard the code from in-depth analysis, a different strategy is imperative, obfuscation. This entails transforming the code so that it retains the same operational semantics, ensuring that it yields identical results when executed but is intentionally rendered significantly more challenging to comprehend. This technique serves the crucial purpose of amplifying the complexity for anyone attempting to dissect the code, whether it be a hacker or a competitor, all while ensuring the code retains its intended functionality. The fundamental objective is to create an effort barrier to dissuade potential adversaries from easily understanding the code's inner workings and thereby enhancing the overall security of the software.

Unfortunately, obfuscation cannot be undefeatable; one way or another, the obfuscated code must eventually execute the same actions as the original code. Consequently, given sufficient time and resources, it can be meticulously analyzed to comprehend its functionalities and operational logic [1]. The objective of an obfuscator, therefore, is not to create an absolutely impenetrable program. Rather, it aims to craft a program

sufficiently intricate and confusing, making analysis substantially more expensive or time-consuming than the potential benefits of deciphering it. In essence, the aim is to raise the bar of effort and complexity to a level where the benefits of the analysis diminish or where, upon successful deobfuscation, the acquired insights are outdated and lost their usefulness.
Numerous code obfuscation techniques are available, each presenting its unique advantages and drawbacks. This thesis centres on investigating a variation of *Just-in-Time* obfuscation, an approach that involves modifying program instructions during runtime, precisely when they are to be executed. This variation involves a continuous process of hiding and restoring segments of the code of a program in a way that only a small part of the code is in memory at all times, intending to increase the challenge for reverse engineers attempting to analyze the program. In doing so, the study aims to find out the viability of the use of this obfuscation method and how it compares with other obfuscation methods in current research.

## 1.1 Research Questions

- How viable is an obfuscation method that continuously restores parts of the code when they are needed, and hides them again after execution?
- How well does this new obfuscation method work regarding Potency, Resilience, Cost and Stealth?
- How does it compare to other similar obfuscation methods?
- Are there any negative security implications of the use of this method for obfuscation?

## 1.2 Thesis Outline

To answer these questions, a Proof of Concept (PoC) of this obfuscation method was designed, implemented and evaluated. The results are compared to the results of other obfuscators in the related literature.
This document is organized in several chapters:
Chapter 2 describes some prerequisites and fundamental knowledge required to understand the study. Chapter 3 lists and comments on the related work found during the literature review. Chapter 4 describes the concept of the proposed obfuscation method, possible weaknesses and additional improvements that could be added to make it more resilient. Chapter 5 details the implementation of the method discussed in the preceding chapter within the PoC. Chapter 6 describes the tests designed to evaluate the program, shows the results, and discusses the performance of the obfuscation method and how it compares to other methods in the literature. Chapter 7 marks the conclusion of the thesis by delving into the results and providing an outlook on potential directions for future research.

# 2 Prerequisites

The thesis touches on various concepts ranging from obfuscation, reverse engineering and executable file formats. This chapter is divided into three sections that explain the required concepts related to each of the topics. Section 2.1 explains executable files, the file formats that define them and how they work, Section 2.4 explains the basics of reverse engineering and the different ways to classify the types of analysis, and section 2.6 gives a basic foundation of obfuscation, presents different common obfuscation methods, and explains the metrics commonly used to evaluate the obfuscators.

## 2.1 Executable files

Computers store programs in the format of executable files that follow a known format that can be understood by the Operating System (OS). These files serve as comprehensive containers that encapsulate all the essential components, code, and resources indispensable for the OS to effectively initiate and execute the associated program.

Upon program execution, the OS parses the file headers to obtain essential data about required resources and memory allocation. Subsequently, it allocates the appropriate memory for the program, proceeds to load the pertinent code and associated resources from the file, and establishes essential memory mappings to the referenced libraries within the program. Lastly, the OS transfers the execution to the entry point of the program.

The formats used by these executable files can vary depending on the OS. Linux systems commonly use the Executable and Linkable Format (ELF), while Windows uses the Portable Executable (PE) format.

## 2.2 ELF File format

In Linux-based operating systems, the designated format employed for executable files is the ELF. This versatile file format also serves additional functions other than executable files, including shared libraries and core files[2].

```c
#define EI_NIDENT 16
        typedef struct {
            unsigned char e_ident[EI_NIDENT];
            uint16_t      e_type;
            uint16_t      e_machine;
            uint32_t      e_version;
            ElfN_Addr     e_entry;
            ElfN_Off      e_phoff;
            ElfN_Off      e_shoff;
            uint32_t      e_flags;
            uint16_t      e_ehsize;
            uint16_t      e_phentsize;
            uint16_t      e_phnum;ma
            uint16_t      e_shentsize;
            uint16_t      e_shnum;
            uint16_t      e_shstrndx;
        } ElfN_Ehdr;
```

Listing 1: Structure of the ELF header

ELF files start with an ELF header, defined in listing 1, which contains a set of bytes that identify an ELF file called magic numbers, processor architecture, endianness, target OS, the type of file, entry point of the program, offsets and other relevant information regarding the program, file and section header.

### 2.2.1 Sections

Defined in the section header (Shdr), sections serve as organizational units for effectively managing and structuring the data within an executable entity, encompassing code, data, and assorted resources instrumental for runtime execution, debugging, or linking.
The Shdr, consists of an array of structures that each encapsulating comprehensive information pertaining to individual sections constituting the program. Each element encompasses key attributes such as the section's designation, classification, flags denoting specific characteristics, file offset, size, alignment requisites, and supplementary details pertinent to the respective section classification.
listing 2 shows the structure used to define sections in the Shdr of 64 bit ELF executables.

### 2.2.2 Segments

Segments continuous blocks of data in the executable file that contain all necessary information necessary to execute the file, like all the code and the data used by it.

```c
typedef struct {
            uint32_t   sh_name;
            uint32_t   sh_type;
            uint64_t   sh_flags;
            Elf64_Addr sh_addr;
            Elf64_Off  sh_offset;
            uint64_t   sh_size;
            uint32_t   sh_link;
            uint32_t   sh_info;
            uint64_t   sh_addralign;
            uint64_t   sh_entsize;
        } Elf64_Shdr;
```

Listing 2: Structure of the Shdr

```c
typedef struct {
            uint32_t   p_type;
            uint32_t   p_flags;
            Elf64_Off  p_offset;
            Elf64_Addr p_vaddr;
            Elf64_Addr p_paddr;
            uint64_t   p_filesz;
            uint64_t   p_memsz;
            uint64_t   p_align;
        } Elf64_Phdr;
```

Listing 3: Structure of the Phdr

The segments are defined in the program header (Phdr), which consists of an array of structures that specify properties of each segment, including its type, size, start position inside the file and parameters that define how the segment is loaded into memory, such as the virtual address, size in memory, flags and alignment. The structure used to store the information about a segment in a 64-bit executable is defined in listing 3.

### 2.2.3 Position Independent Executables

A position-independent executable (PIE) is an executable file format or binary that is designed to be loaded and executed at any memory address, without relying on fixed memory locations [3]. This flexibility allows to implement protections like address space layout randomization (ASLR), where the positions in memory of each part of the program are randomized after each execution[4].

## 2.3 Assembly

The code encapsulated within an executable is not composed in a human-readable programming language but is instead represented in machine code. Machine code constitutes a low-level representation of the code, characterized by hexadecimal values that correspond to instructions executable by the processor. Machine code is tied to the specific architecture of the processor it has been compiled for, and two distinct processor architectures may ascribe entirely different interpretations to identical hexadecimal values.
To understand its meaning, the provided code can be depicted in the form of assembly code, a form of low-level code that is comprehensible to humans and directly corresponds to machine code instructions. This representation serves as an intermediary, facilitating human understanding of machine code instructions.

## 2.4 Reverse Engineering

Rekoff [5] describes Reverse engineering as "...the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system. These specifications are being prepared by persons other than the original designers, without the benefit of any of the original drawings or other documentation". The same principle applies when we talk about Software, where instead of trying to understand how a piece of machinery works, the thing being analysed is a piece of digital data.
The practice of reverse engineering software can serve multiple objectives. A motive involves the analysis of a program to rectify faults, conduct audits, or add new functionalities. In the realm of cybersecurity, this methodology proves instrumental in comprehending the functionality of malign software or ascertaining the security integrity of program code. Malicious entities exploit reverse engineering to disable software protections to enable the illicit utilization of licensed software, identification of software vulnerabilities for the creation of exploits, and the unauthorized acquisition of intellectual property.
We can categorize the techniques employed in reverse engineering a program based on whether the program is executed or not into two primary approaches: Static Analysis and Dynamic Analysis. Additionally, another common method for classifying analysis methods involves distinguishing between Manual and Automated techniques, with each approach requiring or not requiring the presence of an analyst during the process.

### 2.4.1 Basic structures

**Basic Blocks** Allen defines the basic blocks of the program as a "...linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)..." [6], meaning an array of instructions that are always executed in order, starting at the first instruction and executing each instruction until the last without any jumps or loops.

**Control flow graphs** Control flow graphs serve as an effective means for visualizing and analyzing code through their utilization in directed graphs. Each individual basic block within the given code is represented by a distinct node on the graph, while the possible routes that the program can take between these blocks are symbolized by edges connecting corresponding nodes [6]. This representation enables reverse engineers to better understand the analysed programs.

### 2.4.2 Static Analysis

Static analysis is a crucial aspect of understanding software programs at a deeper level. By inspecting machine code without executing the program, analysts can deduce valuable information about the program's structure and algorithms [7]. This process involves employing various tools such as disassemblers and decompilers to gain comprehensive insights into the program's components, relationships, and functionality. It is particularly useful in the context of malware detection or when examining proprietary applications, allowing one to examine the software without risking any potential security threats that may arise from running the code.

### 2.4.3 Dynamic Analysis

Dynamic analysis involves observing a software program while it is executing. This approach allows inspecting the program's runtime behaviour, interactions with the operating system, and interactions with external entities such as files, network resources, or other processes [7].
Dynamic analysis is used to gain insight into the program's actual runtime behaviour, such as how it processes inputs, handles certain conditions, or interacts with external resources. Dynamic analysis is often employed to discover hidden functionality, identify potential bugs, verify hypotheses made during static analysis, or analyze the program's response to different inputs.
Dynamic analysis is a crucial aspect of reverse engineering, focusing on observing and analyzing a program's behaviour while it executes. This method provides valuable insights into various aspects of the program, such as its interactions with the operating system, external entities like files or network resources, and other processes. The primary objective of dynamic analysis is to better understand a software application's actual runtime behaviour, including how it handles different inputs, manages specific conditions, and interacts with external components.

It can also be used to uncover hidden functionalities, detect potential bugs, validate hypotheses generated during static analysis, or examine the program's response to various input parameters.

### 2.4.4 Automated Analysis

Automated analysis involves employing software tools and scripts to dissect executables and extract insights without direct human intervention. Automated analysis is efficient for handling analysis of large amounts of programs that have to be analysed, speed is more important than accuracy, or supporting the analyst for further analysis

Automated analysis involves employing software tools and scripts to examine executable files and derive meaningful insights without relying on direct human intervention. This approach proves advantageous when dealing with large amounts of programs that need to be analyzed, emphasizing efficiency over accuracy, or serving as a support mechanism for further analysis later on.

### 2.4.5 Manual Analysis

Manual analysis requires human expertise to dissect and comprehend the executable's inner workings. Reverse engineers employ their knowledge to figure out how the program works. This approach is more suited when dealing with complex or customized code that automated tools might struggle to interpret accurately. Manual analysis enables reverse engineers to better understand the context and logic of the program, while also allowing them to identify obfuscation and other protections that might trick automated analysis.

### 2.4.6 Symbolic execution

Symbolic execution is a method of reverse engineering that works by executing a program with symbolic inputs, where input values are represented symbolically as variables rather than concrete values. The program is executed by tracking the execution paths and how the variables are modified. This method allows to determine which inputs cause each possible path of the program to be executed. [8]

### 2.4.7 Reverse engineering tools

Reverse engineering tools are programs or applications designed to analyze, decompile, and extract information from existing software programs. These tools are often used to identify vulnerabilities, understand the inner workings of software systems, or even modify existing applications.

**Rizin** Rizin is an open-source reverse engineering framework based on *radare2* that is composed of multiple command line applications that allow to analyse, executable files, visualize their code and structures, and edit and debug them. [9]

**Angr** Angr is a binary platform that allows performing advanced dynamic analysis through the use of symbolic execution. It also includes many preprogrammed analyses that can be used to study the program, like recovering the Control Flow Graph (CFG) of a program to better understand how it works. [10]–[12]

## 2.5 Self-Modifying Code

Self-modifying code refers to a programming technique in which a program alters its own instructions at run time. Because of this, this method can be difficult to understand at first glance, since the instructions that are written on the file are not necessarily the ones that will be executed.

In light of the challenges associated with programming and debugging self-modifying code, its usage has become relatively infrequent in contemporary software development. As a result, numerous reverse engineering tools do not contemplate incorporating support for such code modifications during runtime execution and assume that the underlying code remains unaltered throughout the program's execution. [13]

## 2.6 Code Obfuscation

The main method used to protect software from reverse engineering is code obfuscation. It can be described as the act of transforming code, being source code or machine code, into a harder-to-read equivalent code that maintains the same behaviour. The objective of obfuscation is not to provide absolute security or prevent determined attackers from reverse engineering or tampering with the code., since it is not possible. In the end, the obfuscated program has the same behaviour as the non-obfuscated version and therefore can be analysed and reverse-engineered. Instead, the goal of good software obfuscation is that of increasing the effort required to reverse engineering a program to a point where doing it is more expensive or resource-intensive than the reward of successfully reverse engineering it, acting as a deterrent against analysis attempts.

To decide whether an obfuscation method is good or not, we need a metric that allows us to evaluate it. Collberg *et al.* defines the quality of an obfuscator by the combination of 3 different measures: potency, resilience, and cost.[14]

### 2.6.1 Obfuscation techniques

Many different types of obfuscators can be used to hide the functionality of code.

**Virtualization** works by transforming the original code into a difficult-to-understand intermediate representation and storing it as bytecode. Then, a virtual machine able to read the intermediate representation is used to execute the bytecode generated from the original code. [15]

**Control Flow Flattening** hides the control flow of the program, redirecting all branches of the basic blocks to an intermediate redirects the program to the next basic block, ensuring that the program follows the correct flow of the program. This removes the original control flow of the program, putting all the original blocks next to each other in the graph. [16]

**Opaque Predicates** make code more difficult to understand and analyze by adding conditional statements that have no real impact on the program's behaviour. These statements are designed to be complex and make code appear more complicated to reverse engineers. [17]

**Just-In-Time (JIT) Obfuscation** is an obfuscation method that uses self-modifying code to generate code during runtime when it is needed. Functions are replaced by code that, when executed, uses various methods to reconstruct the original code and execute it. [18]

**Packers** A packer refers to a tool or program used to compress and encrypt executable files, with the objective of reducing its size, and, more importantly, making the executable harder to reverse engineer. Usually, packers compress and encrypt the data of the program, and then add a routine at the start of the program that is responsible for decompressing and decrypting the original code and data into memory before the original code of the program is run.

### 2.6.2 Evaluation metrics

To evaluate how well an obfuscation technique works, four metrics are commonly used: Potency, Resilience, Cost and Stealth. [14], [19] Different methods exist to evaluate each of the metrics,

**Potency** is a measure of the degree of confusion introduced by an obfuscator in a program and how well it performs against manual attacks. The evaluation of potency varies depending on the properties of each specific obfuscator. For those that add instructions such as opaque predicates or modify the control flow graph, potency is typically assessed through a comparison of the complexity between the original and obfuscated code. In contrast, for obfuscators targeting tools used to reverse engineer the code, potency can be quantified by determining how effectively the obfuscation method can mislead these tools into displaying erroneous data.

**Resilience** refers to the capacity of an obfuscator to withstand automated attacks and tools aimed at deciphering its protective measures or extracting information about the original code. This resistance can be quantified by comparing the time needed for an analysis tool to process a piece of obfuscated code, or through assessing the accuracy of the results generated by such an analysis.

**Costs** quantifies the amount of overhead that is added to the program by using the obfuscation technique. It can be derived from the execution time overhead added by using the obfuscation method on a program and by the size increase of the obfuscated files.

**Stealth** assesses how difficult it is to detect that a program has been obfuscated. This concept can be highly specific to each individual obfuscation method, making the stealth metric more theoretical in nature with no standardized empirical tests available for evaluation

# 3 Related Work

This chapter explores the existing body of knowledge related to obfuscation methods and attacks against these methods. The aim is to establish a comprehensive understanding of the current state of research and the advancements made in the field. The literature review was conducted through an analysis of academic journals, and conference proceedings, utilizing carefully crafted queries designed to capture relevant works about obfuscation techniques and the countermeasures employed to breach them. A more detailed list of the queries used can be found here:

- *Obfuscation AND ("self-modifying code" OR "self-modifying code" OR SMC OR "Dynamic obfuscation")*
- *Obfuscation AND (code or program or executable) AND (evaluation OR metric OR metrics)*
- *(deobfuscation OR "reverse engineering" OR "analysis") AND (code OR program OR executable)*

The chapter is structured into two main sections: section 3.2 provides an extensive survey of studies focusing on various obfuscation methods, their types, and their effectiveness. Section 3.1 investigates research concerning attacks aimed at undermining obfuscation efforts, exploring methodologies and strategies employed to compromise obfuscated software. These two focuses of the literature review aim to form a solid foundation for our subsequent research and insights into enhancing the effectiveness and resilience of obfuscation techniques.

## 3.1 Reverse engineering

In this section, we explore related papers in the field of reverse engineering techniques

In [13], Anckaert *et al.* explore a method to calculate and represent CFGs when the program uses self-modifying code, called State-Enhanced Control Flow Graphs. Traditional CFGs do not work well with self-modifying code, since the code changes in run-time, and, by extension, the CFG does too. They propose a new method which calculates all possible CFGs that can appear when the code is modified, and figuring out which blocks of these CFGs can be reached, and then create a new State-Enhanced CFG using all the blocks of these intermediate graphs that can actually be reached.

Dawei *et al.* presents a method to find and potentially deobfuscate programs obfuscated with self-modifying code. By storing the execution trace of the execution of the program and the values on each of the executed memory addresses, the method is able to detect changes on the executed memory and what instructions are actually executed [20]

In terms of analyzing Virtualized code, Rolles explores a step-by-step method to attack code obfuscated using this technique in [15]. the proposed method is a combination of manual and automated analysis to create a compiler that translates the code in the language of the Virtual Machine (VM) into x86 code. It is achieved by finding and reverse engineering the VM to identify what each instruction in the VMs language does and using this information to create a translator to an Intermediate Language (IR). The resulting IR is then optimized and translated into x86. The study then provides an example of this method by creating a deobfuscator for VMProtect, a packer that implements virtualization commonly used by malware.

Salwan *et al.* attempt to create a method able to completely remove the virtualization from the program, obtaining a program equal to or as close as possible to the original.[21] By isolating the code from the VM to the code that it executes, it is possible to reconstruct the original code itself. This is achieved by using Dynamic Taint Analysis in order to separate the instructions from the VM itself from the instructions it is executing, while Symbolic Execution is used to find all execution paths and reconstruct the CFG. This is then combined in new IR that is used to reconstruct the deobfuscated program.

Another study by Kochberger *et al.* provides an overview of the current automatic deobfuscation methods against Virtualization.[22] The paper analyses the different deobfuscation methodologies taking four aspects into account: the amount of information that can be extracted, the effort required to use the method, the degree of automation and the generalization ability. The quality of available virtualization deobfuscation methods is then tested against samples created by various virtualization obfuscators. They conclude that current deobfuscation methods are either really focused on specific obfuscation schemes or require substantial manual work to function.

Shoshitaishvili *et al.* created a binary analysis framework known as *angr* that implements several different analysis techniques described in other papers, along with the possibility of implementing new techniques into it. It works by translating the architecture-specific code of the analyzed binaries into an intermediate representation that is not architecture-specific. The framework also allows performing symbolic execution of the program, recovering the CFG, and even finding vulnerable code and creating exploits for it. [10]–[12]

## 3.2 Obfuscation methods

There are several papers presenting obfuscation methods or aggregating old ones and evaluating them.

[23] proposes a new virtualization obfuscation that aims to fix the weaknesses of current virtualization obfuscators. The new method detaches the method from specific processor architectures and programming languages to be able to obfuscate any binary, and uses new techniques to protect the program against known analysis methods. The method works by creating LLVM IR of the code to obfuscate, which allows compatibility between different architectures. Then, the IR is translated into code of the VM, divided into basic blocks and encrypted with different encryption keys. This encryption is reversed on runtime by the interpreter. The obfuscation method was tested using 5 different samples in terms of strength against deobfuscation attacks, compatibility between architectures and overhead created by the obfuscation. The samples obfuscated with this method showed a significant increase in terms of strength against deobfuscation compared to the Tigress virtualization obfuscator but at a high computational cost.

In [24], Schrittwieser *et al.* implement an obfuscation scheme that focuses on making dynamic reverse engineering attacks harder by combining several obfuscation methods, each targeting a different component of the reverse engineering process, while also introducing the concept of diversification into code obfuscation. By creating multiple paths that the program can take depending on its input, the complexity of an attack against this obfuscation scheme is greatly increased. The potency, resilience and cost of the obfuscation scheme are then evaluated using two programs: an AES implementation and a benchmarking tool. To evaluate the cost, the original and obfuscated versions of the programs were compared regarding their performance and size. The potency was determined by comparing software complexity metrics, specifically the Length, Nesting Complexity and Data Flow Complexity. Last of all, resilience was evaluated by calculating the amount of paths that would have to be tested to analyze a program after obfuscation, along with trying to reconstruct the original code using two reverse engineering tools.

Pawlowski *et al.* present a method that follows a similar approach in [25]. Their new obfuscation technique introduces a probabilistic control flow. Programs obfuscated with this method do not follow the same path between two executions even when using the same input, while achieving the same end result. This method can help protect the program against dynamic analysis, while also making static analysis harder. The performance metrics of this method were calculated in various ways, using a program that implements SHA-256 and its obfuscated version. The cost of was evaluated by comparing the size, execution time and memory consumption of both programs. The resilience was calculated by comparing the similarity of the traces made by different executions of the obfuscated SHA-256 program with the same input. The potency was measured by retrieving a trace made by the original program and the obfuscated version, adding up the number of basic blocks that are executed during a run of the program, and showing that the amount of basic blocks that are visited during execution is drastically increased in the obfuscated version, and thus, there is more code to analyze. Lastly, stealth was discussed theoretically.

Ebad *et al.* conducts a systematic literature review (SLR) [26] about the ways obfuscation methods are measured. They compare how obfuscation methods are evaluated. In their study, they look at methodologies used for testing, datasets, attributes used to measure the obfuscators (Potency, resilience, cost, stealth and similarity), and how these attributes are measured in each of the studies. They also point out how there is a lack of proper methods to objectively evaluate stealth. On the topic of self-modifying code, Balachandran *et al.* propose a new method to obfuscate code using self-modifying code on [27]. Their method works by hiding information about instructions that modify the control flow of the program (jumps) in the data area, using this information to reconstruct said jumps when needed, and using it to hide the instruction after it is executed. The authors measured the performance of the obfuscation using Potency, Cost and Stealth using programs from the SPECint-2006 framework.The Potency against static analysis was calculated by analyzing the programs with IDA Pro and comparing the result with the one from the original binaries. The Potency against dynamic analysis was calculated by running the program and pausing execution on random locations, disassembling it and searching for differences between each pause, to try to find the deobfuscated instructions in memory. Then, they calculated how many tries they needed to do to find each obfuscated jump instruction using this method, and the time consumed by this analysis. The Cost was evaluated by measuring the increase in program size and execution time in the obfuscated programs. Lastly, the authors used the Mahalanobis distance between the original programs and the obfuscated programs in order to calculate the Stealth of the obfuscator.

Another obfuscation scheme based on self-modifying code is described in [28]. The proposed theoretical method combines irreducible loops and self-modifying code in order to create harder-to-crack binaries. Since the proposal is theoretical Morse *et al.* does not provide metrics for the cost, potency, resilience and stealth of the program, but does some theoretical analysis of the possible costs of the obfuscation along with the difficulty of an automated attack.

Kanzaki *et al.* present a method of JIT obfuscation [29] that replaces a number of instructions of the program with other *dummy* instructions, then, code is added in places before and after the execution path of each *dummy* instruction that replaces them with the original, restoring the semantics of the program, and then changes them back to their hidden value. The paper evaluates the size overhead of their method by obfuscating the program *gzip* with multiple configurations, changing the number of obfuscated instructions.

The performance overhead is measured by running the obfuscated programs to compress a 1Mb text file with the obfuscated programs.

## 3.3 Discussion

The current body of literature heavily emphasizes virtualization as a primary focus in both reverse engineering and obfuscation. Emerging research continues to evolve, pushing towards the development of increasingly sophisticated attack methods that approach a fully automated deobfuscation process, though this objective has not yet been achieved. Concurrently, efforts to fortify virtualization against analysis and hinder deobfuscation are recurrent themes within recent literature.

While virtualization dominates research attention alternative obfuscation methods also appear, although not as commonly, in current studies. These methods primarily target specific automated attacks, substantially elongating the time required for their execution, or thwarting them altogether. However, their effectiveness against manual or assisted analysis remains uncertain.

A noticeable research gap pertains to self-modifying code in obfuscation, encompassing both the scarcity of studies on obfuscation techniques involving self-modifying code and the sparseness of research on corresponding attack methodologies. Although tools like "*angr* purposely support self-modifying code, their use while analysing it is limited, highlighting the need for further exploration in this domain.

Regarding method evaluation, a significant number of studies of obfuscation techniques lack variety and quantity in the datasets used to evaluate the obfuscation method. This limitation compromises the validity of findings, as small sample sizes may introduce bias due to the possible considerable variation among programs.

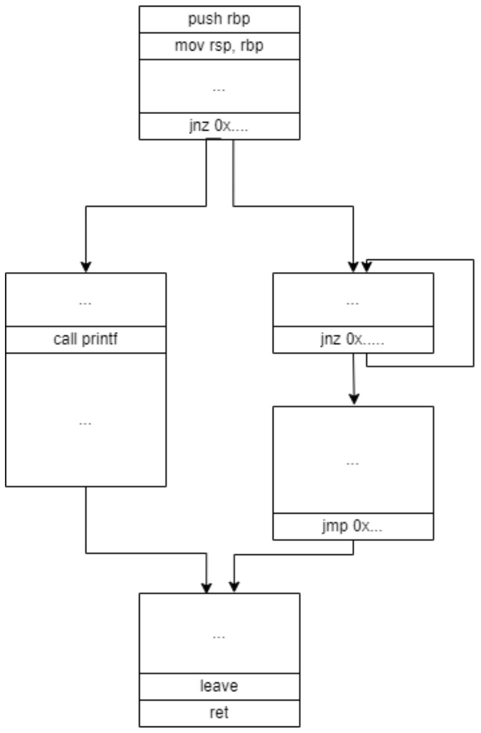
# 4 Concept

A problem with common executable packers is that, even though the original code is hidden at the start of execution after the unpacking function is executed, the original code of the program can easily be retrieved from memory. That is possible because most packers restore the whole executable at once before the instruction pointer is redirected to the original entry point of the program. [15] Common well-known packers like UPX [30] and Upack/WinUpack, two of the most commonly used packers in malware [31], can be easily unpacked this way by running the program until the part of the unpacking function that jumps to the location where the original will be unpacked. At that point, the original executable is complete in memory and can be dumped to a file [32]–[34].

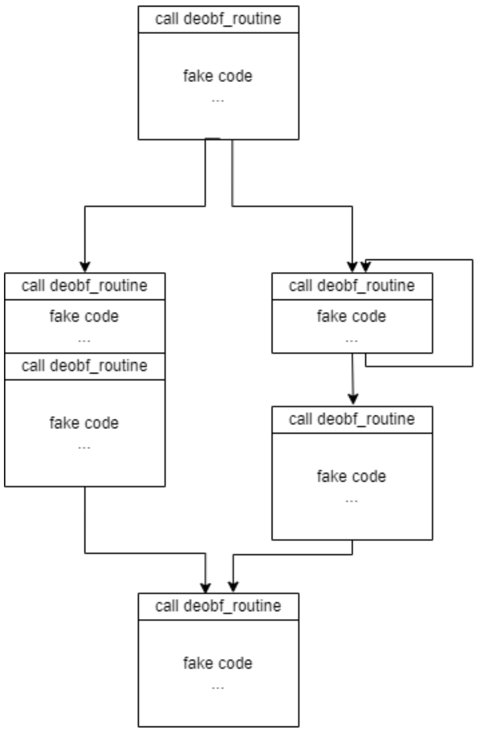
Attacks against Virtualization have been a focus of research [15], [21], [22] for years, and even though they are not perfect yet and require some improvement before they can be used in a fully automated way, are becoming increasingly effective at recovering the original CFG of programs, or even a good approximation of the original code that can work without the interpreter. Additionally, virtualization is also very computationally expensive compared to other obfuscation methods. These factors can make virtualization less suitable for specific applications.

A small amount of research exists regarding self-modifying code compared to other obfuscation methods. This lack of attention is not only shown in literature about the creation and evaluation of obfuscation methods, but also in the reverse engineering and analysis research, and in analysis tools, which usually ignore it, not taking into account that the code might not be static during the whole execution. Furthermore, self-modifying code is confusing to read, understand and debug, which makes manual analysis notoriously hard too.[13] These factors work in favour of obfuscation techniques that use self-modifying code, which will go undetected by most commonly used tools and will require significant effort to be detected and analysed.

The objective of this research is to create a JIT obfuscator that will continuously restore and hide the basic blocks when they need to be executed. The basic concept of the method is to divide each function into basic blocks and transform the contents into unknown data. Before each basic block is executed, a function will restore the original contents of the basic block and hide the previously executed one. The first instruction



(a) Simplified representation of the CFG of a function.



(b) Contents of the function after obfuscation.

Figure 4.1: Representation of the effects of the obfuscation on the assembly code of the program.

of each identified basic block will be replaced with a call to a new function. This function will restore the block that has to be executed next, hide the previously executed block, and then return the execution to the original instruction. The rest of the bytes of each block will be replaced with fake code that will also contain information necessary to restore the block.

Figure 4.1a shows a simplified representation of the CFG of a function in assembly, Figure 4.1b shows the same function after applying the obfuscation method, replacing all the original code with fake code and calls to the deobfuscation function. The obfuscation replaced the code that controls the execution flow of the program, found at the end of the blocks, preventing the reconstruction of the original CFG of the program by inspecting the code of the function using static analysis.

This way, the computer only has a small part of the code of the program in memory at a time, while the rest remains obfuscated. In the meantime, when executed, the semantics of the program remain unchanged.

There exist some similarities between this obfuscation method and the ones mentioned in chapter 3. The obfuscation method [27] detailed by Balachandran *et al.* also uses self-modifying code to restore the semantics of individual blocks on runtime. Their method works by swapping the jump instructions for *mov* instructions, which are then reconstructed on runtime by a new block of code embedded just before the obfuscated instruction, and re-obfuscated soon after execution, furthermore, junk bytes are inserted on the code that confuses disassemblers and makes them think the instructions start on the wrong byte. Even though this method also hides the control flow of the program, it only hides the hex data of the jump instructions, and the rest of the code remains unchanged.

Similarly, the method [29] described by Kanzaki *et al.* also hides single instructions which are also restored during runtime and re-obfuscated after they have been executed. This method does not hide all the instructions of the program and thus might reveal crucial parts of the original code.

Another obfuscation method in the literature [23], proposes a virtualization obfuscator that, after transforming the code of the program to code of the VM, divides it into basic blocks and encrypts them with a key, then embedded on the header of the block, which is then used by the interpreter in runtime to decrypt the VM instructions when needed. Similar to how the method proposed in this study works, this method hides each block separately, and restores them at runtime when they need to be executed. However, the added overhead from having to decrypt the code from all blocks along with the, already high, inherent overhead of virtualization obfuscators makes this obfuscation method only suitable for extremely sensitive functions where speed is not a concern.

## 4.1 Challenges

To achieve this we need to face the following challenges:

- The sections and segments with the code are usually not editable during runtime.
- A function and other required information need to be stored somewhere.
- All basic blocks of the relevant functions must be found.
- An algorithm to hide and restore the relevant code.
- A way to return the execution to the start of the basic block.
- Identify which block is calling the function and has to be restored.
- Find which block was previously restored to hide it again.

The first problem can easily fixed by modifying the p_flags in the Phdr and the *sh_flags* in the Shdr to allow the program to modify that section on runtime. Another way to do it would be to restore the program to a new section or an existing one with enough unused space. This second method would require accounting for the new offset in memory of the restored code, which would affect parts of the code where the memory addresses are relevant.

To store the new function and all relevant data, new sections can be created by modifying the ELF header to indicate that there are more entries on the Shdr, and add structures to it with the information of this new entry. The new sections could then be used to store code and data. Another option for the function would be to use code caves, an array of unused bytes in a program that are loaded in memory and can be used to inject instructions into the program, in order to store the new function in memory. Depending on the method used to hide the basic blocks inside the program, the data that would have to be stored could be small enough to be stored in unused parts of the memory of the program.

The code blocks can be found by identifying all the instructions that change the course of the program (jmp, jnz, jz, etc.), its possible destinations, the entry point of a function and its end. All blocks start at a potential destination of one of these instructions or at the entry point of the function. Then, the end of each block will be determined by the closest start of the next block, or by the end of the function. Even though usually basic code blocks do not take function calls into account, they also will be used to mark the end of a code block and the start of the next for our purpose. Without it, a call to another function that is also obfuscated would return to the middle of an obfuscated block, since the called function would obfuscate it while restoring its first block. The blocks found using the previous method that contain a call will then be divided in two, one that ends on the call instruction and another that starts in the instruction after the call.

Regarding the algorithms, multiple possibilities could be used, each with its advantages and disadvantages. Modern ciphers could be used, but they would be computationally expensive, another option could be to use XOR with a set mask through all the hidden bytes of the block, which would be extremely fast but would make it very easy to recover. Both of these methods wold require a different method to restore the first 5 bytes of each block because we don't have full control over them since they have to be used for the call to the function to restore the block. Another possible method would be to xor the block with a string of data we store somewhere else in the program. This way gives full control of what are the contents of the modified block before restoring it but requires extra storage space and would probably be slightly more computationally expensive than *XORing* it with a set value each time.

If nothing is changed in the function that restores the code, when it returns execution to the deobfuscated code it would execute the instruction that starts after the original call instruction, skipping the first 5 bytes of the basic block and landing on the wrong instruction or even an invalid one. The de-obfuscation function has to modify the return address on the stack to ensure that it points to the start of the basic block and the code is executed in the same order as in the original file.

To know which is the block that is calling the function to restore itself, the memory address of the return address can be used to find where the code has to be restored, but it cannot be used to identify which of the blocks of the program is because, since PIE randomizes the memory positions each execution, each time

the program is run this value will be different. Instead, the value of the five bytes that form the call could be used. They are not modified when loading the program, and since CALL instructions point to an offset in memory from their position, each call to the same function will have a different hex value in memory. Another possibility, it would be possible to use calls to different destinations for each block, and have code in each of these destinations that calculates an identifier used by the function to know from which block the call came from.
For the last point, the information required to identify which was the last block that was just executed and needs to be hidden again, which are the position in memory and the data used to identify the block, could be stored in a fixed position in either an unused space in the memory of the program or in a reserved place in the section used to store the data used to restore the blocks.

## 4.2 Benefits

This method to obfuscate the program has some benefits from commonly used methods to pack and obfuscate code.
First, the original code can no longer be recovered by just executing the program until it finishes the unpacking process, since the obfuscated program is continuously modified, hiding already executed basic blocks and restoring the ones that are going to be executed next.
Since software breakpoints modify the code in memory, they will break when the program modifies itself, or even prevent the restoring function from working correctly. So it will limit the capabilities of debugging the program and forcing the use of hardware breakpoints, which are more limited since only a certain amount of hardware breakpoints can be used at once.
Furthermore, depending on the method used to hide and restore the original code, we can place fake code in place of the original functions that might be able to trick some automated tools and analysts into thinking that that is the actual code that will be executed.
Another benefit of this method is that since code that modifies itself is not commonly studied and taken into account in automatic tools, it can prevent them from working correctly or even make them malfunction.[13]

## 4.3 Weaknesses

Some potential weaknesses have been identified that could be used to attack this obfuscation method:
The start of the basic blocks can be easily identified with the calls to the function used to hide and restore the basic blocks, and with enough knowledge about this function, one could identify the calls to the obfuscation

method and use this information to restore each of the blocks one by one and restoring the full function.

Another possible attack vector against this packer would be to run the program and save the restored blocks one by one until the whole program is restored. Since not every block might be executed on a single execution of the program, other methods like symbolic execution would need to be used to find paths to all blocks and fully deobfuscate the program.

Also, blocks smaller than five bytes in size cannot be replaced by a call since it would require writing over the next block, so they have to stay in their original form. This does not have a big effect on the obfuscation since blocks with 4 or less bytes usually consist of just one or two instructions, revealing little of what the program does or how it does it.

Jumps where the destination is calculated during run-time can be difficult to predict, and if not handled properly might not work properly with this method, leading to crashes or unexpected results.

This method only obfuscates the code and not the other data used by the program. If obfuscating this data is necessary for the purpose of obfuscating an executable, another obfuscation method should be used along this one to hide this data too.

The original code of the program is executed between calls to the deobfuscator, this means that the concepts behind modern attacks against virtualization, like symbolic deobfuscation and taint analysis [20], [21], might be adaptable to attack this method and automatically separate the original code of the program from the deobfuscation function and recover the original code.

Lastly, another weakness of this method, although not related to obfuscation, is that it requires the sections of the program that contain code to be writeable, which could make them more susceptible to exploitation if a bug exists in the code that introduces a vulnerability.

## 4.4 Countermeasures

Some possible countermeasures would increase the resistance against possible attacks mentioned in section 4.3.

In order to make the obfuscator more resilient against the program being deobfuscated by finding the blocks and deobfuscating them, two countermeasures are proposed. The first consists of hiding the calls from the assembly code that can be retrieved from static analysis tools. By setting the bytes previous to the call instruction to bytes that are the start of a long opcode, we can make static analysis tools read the bytes that form the call as part of the previous instruction, effectively hiding the call from them. listing 4 shows an example assembly code with a call to the obfuscation routine, if the two bytes before the call are replaced

```
nop                              ; 90
nop                              ; 90
call   sym.function              ; e8 a1 ae 00 00
nop                              ; 90
nop                              ; 90
nop                              ; 90
```

Listing 4: Example call to the obfuscation routine

```
movabs rax, 0x9090900000aea1  ; 48 b8 e8 a1 ae 00 00 90 90 90
```

Listing 5: Modified hidden call to the obfuscation routine

by *0x48b8*, these bytes represent the opcode for *movabs* that stores the 8 bytes that follow to the *rax* register. Since the bytes that follow are the ones that form the call, they will not be shown in the assembly code by the disassembler and will be part of the *mov* instruction, as seen in listing 5.

The second proposed countermeasure tries to hide the location of the real blocks by adding fake calls to the deobfuscation function in the middle of real blocks and fake data in the data section. The existence of fake blocks that overlap with the real ones helps harden the obfuscation against deobfuscation attempts since to know if a block is reachable or not would require the program to be executed, or at least would require an analysis that traces all possible paths of the program to identify which blocks can be reached and therefore are real which ones cannot.

It is also possible to add calls to the transformation function that restores fake code that is actually executed, which then in turn would call the deobfuscation function, restoring and executing the original code of the program. Using these fake blocks would not only safeguard the obfuscated program by the attacks described in the previous paragraph but would also reduce the effectiveness of attacks that trace the executed instructions as the one described at [20].

This could be further improved by using this obfuscation method along obfuscation methods that increase the possible execution paths of the program like the ones shown in [24] or [25]. This would greatly increase the number of paths that would need to be analysed in order to find the real blocks and to deobfuscate the whole program.

# 5 Implementation

Knowing all the requirements and challenges of the concept explained in chapter 4, we can implement a PoC that will allow to test its performance as an obfuscator. The packer will be written in Python since it has access to many libraries relevant to this project.

For this PoC, the following design choices were made to face the challenges discussed in chapter 4:

The packer will be implemented using new sections to store the function and all the required data. This will allow the packer to work with more programs since it will not matter if there is enough unused space, which in turn will allow more flexibility when testing.

The flags of the sections that contain the functions to obfuscate will be modified to be able to edit them on runtime.

For the algorithm used to hide and restore the original code, each basic block will have a corresponding hex array of the same length stored in the new data section that will be XORed with it. This allows the use of the same function for hiding and restoring the original code while allowing full control of the fake code that is shown in place of the original.

To obfuscate and deobfuscate the basic blocks, the first five bytes of each block will be replaced with a call to a function that will fetch an identifier and a memory offset of the last restored block stored on the custom data section, and hide it again. Then, the first five bytes on the address referenced by the return address to identify the block, restore it, and store this information on the data section in order to be able to obfuscate it again on the next call.

Additionally, the obfuscator adds fake calls to the deobfuscation function and their respective data in the data section to emulate a basic block that will never execute. These fake blocks will add confusion to the obfuscated program since it will not be possible to figure out if a block is real until the program is executed and an execution path that leads to it is found.

Lastly, the bytes previous to a call to the function to restore the code are replaced by the first bytes of another instruction. This will trick disassembly tools into using these bytes to identify the instruction and use the bytes that form the call instruction as part of the previous instruction, effectively hiding the calls from the disassembled instructions.

## 5.1 Technical Requirements

To create the packer, we need a way to parse the ELF file and all of its properties, find and analyse the relevant functions and edit the headers, code and data.
In order to find and edit information about the executable and its properties, like sections and segments, the Python library for the Library to Instrument Executable Formats (LIEF) Project will be used. The Python library rzpipe, an API for rizin, can help with the analysis of the machine code in the executable, while also allowing editing the code and hex data of the executable.

**LIEF** The LIEF project provides libraries that allow parsing multiple executable file formats, including ELF and modifying some of its parameters.[35] This allows to get all the necessary information from the Shdr and Phdr, and modify them to make the necessary sections writeable and create the new sections that will store the function to restore and hide blocks, and the data needed for it to work.

**Rizin** Rizin [9] is a command line reverse engineering framework that provides an API like scripting engine through the library rzpipe. Using it allows one to get the assembly code of the program, functions, basic blocks and references, which is all the required information to apply the obfuscation into the program. Rizin also allows to edit the contents of the file. Writing hex data or inserting custom code inside the executable. This functionality will allow to make all the changes in the executable file to hide the original code, add the transformation function, and add all the data it needs in order to restore the original code.

## 5.2 Overview

In Figure 5.1 we can see a representation of the steps the obfuscator follows while transforming an executable to add this method.
The obfuscator follows 6 steps, where it analyzes and transforms the original program. First, the program is analysed to identify the size and location of the relevant functions. Then, the headers of the file are modified to make the sections that contain the functions we want to obfuscate writeable during runtime and to add the sections that will contain the additional data and code used by the obfuscation method. The file with the new sections is then analysed to identify all the basic blocks of the functions that will be obfuscated. These are modified, adding the calls to the transformation function at the start of each block, while the rest of each

basic block is replaced by the bogus code. Once the original functions are hidden, the xor data required to restore them is stored in the data section, along with the required information to find the size of the data of each block and where exactly it is located in the section. Lastly, the code that hides and restores each block is stored in the section created to store it.

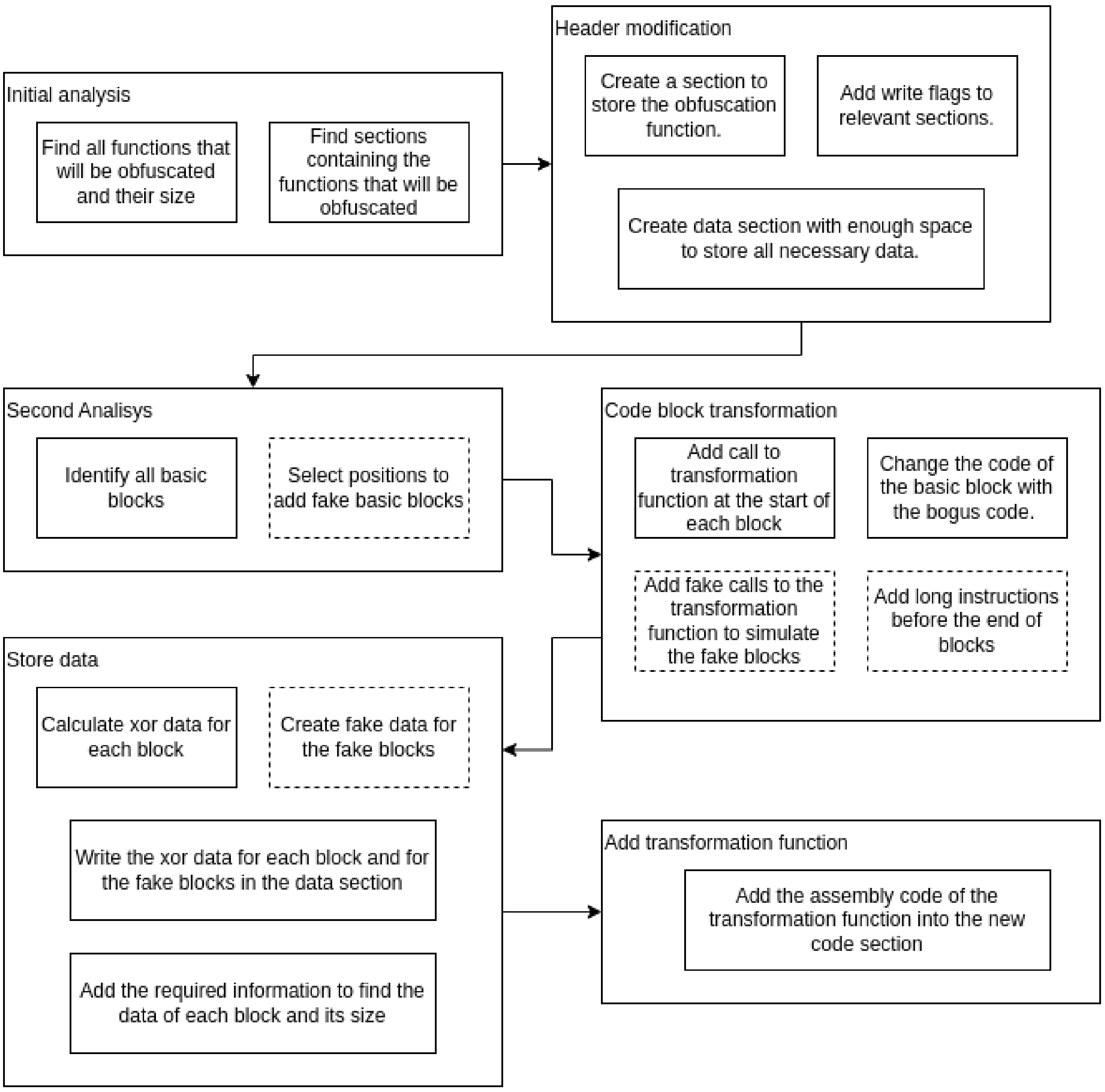

Initial analysis
Find all functions that will be obfuscated and their size
Find sections containing the functions that will be obfuscated
Header modification
Create a section to store the obfuscation function.
Add write flags to relevant sections.
Create data section with enough space to store all necessary data.
Second Analisys
Identify all basic blocks
Select positions to add fake basic blocks
Code block transformation
Add call to transformation function at the start of each block
Change the code of the basic block with the bogus code.
Add fake calls to the transformation function to simulate the fake blocks
Add long instructions before the end of blocks
Store data
Calculate xor data for each block
Create fake data for the fake blocks
Write the xor data for each block and for the fake blocks in the data section
Add the required information to find the data of each block and its size
Add transformation function
Add the assembly code of the transformation function into the new code section

Figure 5.1: Overview of the implementation of the obfuscator.

## 5.3 Initial analysis

The first step before starting to modify the program is to identify where the functions that want to be obfuscated are located, and their size. The location will allow to find which sections we have to modify to allow rewriting the functions during the execution, while the size is needed to calculate how long the new data section has to be.
Before anything else, the program is analysed with the rizin *aaa* command, which performs an initial analysis of the program. Rizin describes the use of this command as "*Analyze all calls, references, emulation and applies signatures*". This is necessary for the rest of the rizin functionalities to work.
The information about the function size and their location is acquired through the rizin command *aflj*, which returns a list of all functions, their location, size, and other interesting information.
The location can then be used along the command *iS*, which lists information about the sections of the executable, to get the sections of the functions.

## 5.4 Header modification

With the necessary information about the functions and sections, it is time to modify the headers of the ELF file.
Using Lief, we can open the executable file and modify the flags of each header.
To enable write access in the required sections the flag *lief.ELF.SECTION_FLAGS.WRITE.value* has to be set, in the same manner, the flag *lief.ELF.SEGMENT_FLAGS.W* has to be set on the segment.
LIEF also allows creating the new sections. The code used to create a new section can be found on listing 6. The type used for the new sections is *SHT_PROGBITS*, which is defined as a section whose contents are defined by the program and, therefore does not have to follow any predetermined format.[2] The *SHF_ALLOC*, *SHF_WRITE* and *SHF_EXECINSTR* can be set here before adding the new section to the section header. All necessary changes to the ELF header to add the new sections to the program are automatically handled by LIEF.
After all changes have been set, a copy of the program with the modified headers is saved.

## 5.5 Second analysis

The new modified program can be opened with rizin and analysed with *aaa*. Then, the basic blocks can be found using the commands *afb*, which returns a list of the basic blocks of the program, and *afx*, which lists

```python
def add_section(self, section_name, size, exec=True):
        ...
        section.type = lief.ELF.SECTION_TYPES.PROGBITS
        ...
        section.content = [0x00]*size
        ...
        section.flags |= lief.ELF.SECTION_FLAGS.ALLOC.value
        if exec:
            section.flags |= lief.ELF.SECTION_FLAGS.EXECINSTR.value
        else:
            section.flags |= lief.ELF.SECTION_FLAGS.WRITE.value
        self.target.add(section)
```

Listing 6: Function to add a new section

references from and to the function. With this information is possible to find all basic blocks and divide them if they have a call to another function or to itself inside, to avoid the problems stated in section 4.1, where calls to other obfuscated functions would return to obfuscated code and not execute properly.

The blocks smaller than 5 bytes can then be discarded, while the rest will be obfuscated. Furthermore, blocks bigger than 10 bytes will be identified to be used to add fake calls to the transformation function later.

## 5.6 Code block transformation

During this step, the contents of each block are transformed to hide the actual code to be executed.

First, the first 5 bytes of each block are changed to a call to the function that will restore it. This can be done with the *wa* rizin command, which accepts assembly code as input, translates it to machine code, and writes it into the program.

Then, the rest of the code of the block is replaced by the bogus code, including calls to the transformation function that will never execute.

To write the first bytes of the long *mov* instructions that will hide the calls at the start of the following block, the last two bytes are changed with rizin's *wx* instruction, which allows to write hex data into the program.

## 5.7 Store data

Now the original code is no longer on the program, but some extra data is needed to be able to restore it, meaning, the XOR data to restore each block, and a way to find it.

The section created to store the data starts with two 8-byte fields used by the transformation function to store the location in memory of the last restored block and its identifier, an ID unique to each of the blocks. The following bytes contain an array of structures, one for each block, containing the ID of the block, the length of the XOR data, and the location of the data. Then, the XOR data of all the blocks is stored after the array. A representation of the section can be found in fig. 5.2.

To read all the hex contents of each block, the rizin command *px* is used, which returns the hex values of the data stored on a given location in the program.

The obfuscator calculates the XOR data to restore each of the blocks by XORing the original block with the modified block. Then, the ID is extracted from the first 5 bytes of each block, which corresponds to the call to the transformation function, and it is sure to be different in each block. The option to use more than five bytes was contemplated, but, because of PIE and ASLR, in certain conditions, the data after the last block of a function would change each execution and make the stored ID differ from the ID during execution. All write operations to the data section are performed with the command *wx* from rizin,

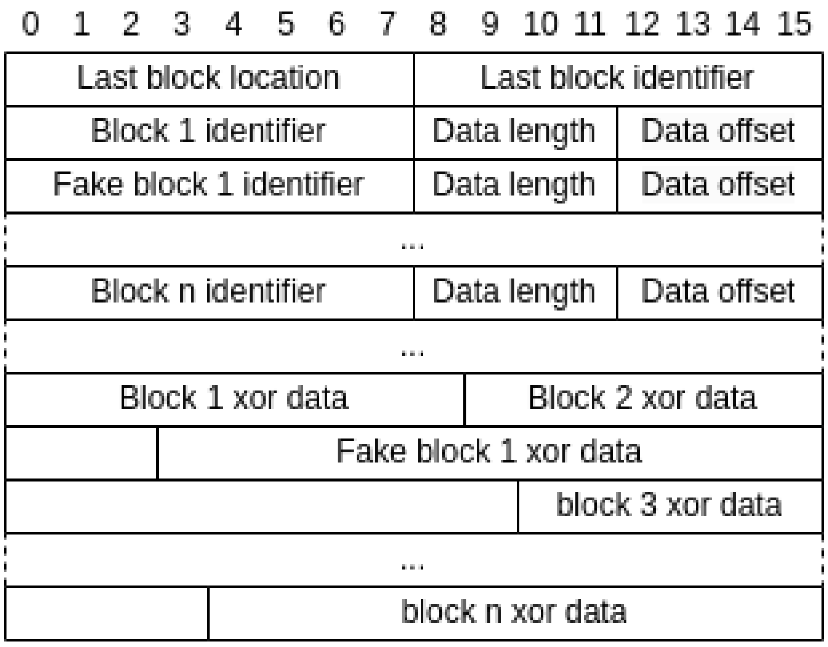


Figure 5.2: Structure of the data section.

```
...
sub    qword [rbp+40], 5                 ; Change return address to
  ↪  the start of the block.
lea    rdx, .data_section
mov    rax, qword [rdx]        ; Load previous block location
test   rax, rax                                 ; If first execution,
  ↪  skip
jz     .skip
mov    rbx, qword [rdx + 8]    ; Load previous identifier
call   .xor_block              ; Hide previous block
.skip
lea    rax, [rbp+40]
mov    rax, qword [rax]
movabs rbx, 0xffffffffff
and    rbx, qword [rax]
mov    qword [rdx], rax        ; Store current block address
mov    qword [rdx + 8], rbx    ; Store current block identifier
call   .xor_block              ; Restore current block
...
```

Listing 7: Code used to find the old and new IDs and call the xor code.

## 5.8 Add transformation function

After these changes, the code of the obfuscated functions is hidden, but, executing it in this state will lead to incorrect results, and in some cases, a crash. To restore the semantics of the program and ensure that it can be executed as intended the code that will hide and restore blocks has to be inserted into the program. This code is formed by two small assembly functions; the first, shown in listing 8, finds the identifiers and locations of the blocks; the second, shown in listing 7, searches for the data of the block in the data section and uses it with XOR to restore or hide the block.

To write the code in the new code section, the rizin *wa* command is used.

## 5.9 Additional functionalities

The obfuscator contains multiple additional functionalities to help with the testing, which allows to control which blocks and functions are obfuscated or activate additional protections.

To be able to test all the programs in chapter 6 without manually identifying and selecting functions to obfuscate on all the programs, an option allows one to automatically search and obfuscate all internal functions

```
.xor_block
mov    rcx, rdx
.id_check
add    rcx, 0x10
cmp    rbx, qword [rcx]                    ; Check the identificator
jne    .id_check
push   rdx
push   rax
mov    rbx, qword [rcx + 8]
mov    rcx, qword [rcx + 0xc]
movabs rax, 0xffffffff
and    rbx, rax
and    rcx, rax
add    rcx, rdx
pop    rax
.xor_block
mov    rdx, qword [rcx]
xor    qword [rax], rdx          ; XOR current location with the stored
 ↪  value on the data section.
add    rcx, 8
add    rax, 8
sub    rbx, 8
cmp    rbx, 0
jg     .xor_block                ; Repeat until the whole block is
 ↪  restored.
...
```

Listing 8: XOR function

of the program while skipping imported functions.

To check how the different parts of the obfuscation functions and the sizes of the blocks affect the execution time, options were added that allow limiting the amount of bytes that are obfuscated for each block, or only obfuscate blocks of a specific size range.

Last, an option allows to activate the two countermeasures described in section 4.4, hiding the calls to the transformation function behind *mov* instructions of 64-bit values and adding fake calls to the program.

# 6 Evaluation

This section evaluates the potency, resilience, cost and stealth of the obfuscation method using a data set of 83 programs compiled in 18 different ways each, with a total of 1494 executable files.
The obfuscation method is evaluated using several metrics and tests that follow current trends in the literature. The cost of the obfuscation method is gauged by measuring the execution time overhead using *Hyperfine* [36]. Rizin [9] is used to check how resilient the obfuscation method is against Static Analysis tools, and is also used to discuss the potency of the obfuscation, *Angr* [10]–[12] is used to evaluate the method against automated dynamic analysis attacks. Other literature had problems finding a reliable approach to test the stealth of obfuscation methods, and there is no consensus on how to do it, so the stealth of the obfuscator is done theoretically.
section 6.1 explains how the tests are designed and executed, and then shows the raw data. section 6.4 interprets the data and correlates it with the **potency**, **resilience**, **cost** and **stealth** metrics of the obfuscation method.

## 6.1 Data Collection

A machine running Ubuntu 22.04.2 LTS with an *Intel(R) Xeon(R) CPU E5-2650 v4* and 4 GB of RAM was used for the tests.
The dataset used to perform the tests consists of 83 different programs written in C language, which were used to create the executable files for testing. Each program was compiled with 4 compilers: GCC, clang, *compcert* and *tinycc*. For *clang* and *gcc*, their 4 optimizations were used to create 4 different executable files each. All used compilation methods are listed below:

- Clang: Optimization level 0
- Clang: Optimization level 1
- Clang: Optimization level 2
- Clang: Optimization level 3
- Gcc: Optimization level 0

| Compilation | Size avg. | 0-8 B | 8-16B | 16-32B | 32-64B | 64B |
|---|---|---|---|---|---|---|
| **All** | 19.36 | 33.49% | 33.60% | 22.71% | 6.69% | 3.50% |
| **Clang** | 15.61 | 34.48% | 33.03% | 23.69% | 6.96% | 1.84% |
| **Gcc** | 17.50 | 33.84% | 34.71% | 20.97% | 6.12% | 4.36% |
| **Compcertcc** | 67.44 | 31.97% | 28.64% | 32.41% | 6.16% | 0.81% |
| **Tinycc** | 22.35 | 21.50% | 23.07% | 36.31% | 16.25% | 2.87% |
| **Flatten** | 15.66 | 28.78% | 39.29% | 23.23% | 7.35% | 1.36% |
| **Virtualize** | 35.64 | 50.36% | 11.38% | 10.63% | 0.50% | 27.14% |

Table 6.1: Information about basic blocks for each compilation method.

- Gcc: Optimization level 1
- Gcc: Optimization level 2
- Gcc: Optimization level 3
- Compcertcc: Default configuration
- Tinycc: Default configuration

Additionally, the Flatten and the Virtualize transformations of the Tigress obfuscator [37] were used separately to create two obfuscated versions of each program. These programs were compiled 4 times with the 4 default optimization levels of GCC. In total, each program had 18 variations, leading to 1494 executable files, 50 of which had to be discarded because they failed to execute properly, leaving 1444 programs to test. Before obfuscation, the basic blocks of the programs were analysed, comparing average size, and the amount of blocks of different size ranges. Table 6.1 displays information about the basic blocs of the programs used during the evaluation.

To evaluate potential variations of this obfuscation method, the programs are obfuscated using 7 different configurations of the packer to see how different parts of the obfuscation affect the program:

- *Conf 1:* Default obfuscation of all blocks.
- *Conf. 2:* Obfuscation of the first 8 bytes of each block.
- *Conf. 3:* Obfuscation of the first 16 bytes of each block.
- *Conf. 4:* Obfuscation of the first 32 bytes of each block.
- *Conf. 5:* Obfuscation of blocks bigger than 16 bytes.
- *Conf. 6:* Obfuscation of blocks bigger than 32 bytes.
- *Conf. 7:* Obfuscation of blocks bigger than 64 bytes.

| Compilation | Original files (number) | Obfuscated |
|---|---|---|
| **All** | 1444 | 86.75% |
| **Clang** | 328 | 92.45% |
| **compcertcc** | 64 | 98.21% |
| **gcc** | 328 | 96.16% |
| **tinycc** | 68 | 25.00% |
| **Flatten** | 328 | 87.19% |
| **Virtualize** | 328 | 81.75% |

Table 6.2: Successfully obfuscated files

The first obfuscates all blocks that can be obfuscated taking into account the obfuscator limitation of 5 bytes per block. Configurations 2 to 4 explore how the obfuscation performs when only the first 8, 16 and 32 bytes of each block are obfuscated, leaving the rest of the block untouched. The tests are designed to use multiples of 8 because of how the loop that obfuscates each block hides/restores 8 bytes at a time, so numbers in between do not affect the number of iterations the loop has to go through. Configurations 5, 6 and 7 explore how the obfuscation method performs when only blocks bigger than 16, 32 and 64 bytes respectively are obfuscated. As seen in the table 6.1, the distribution of the sizes of blocks varies greatly depending on the compilation method, which might affect the results of the different configurations.

Since manually selecting the functions on each of the programs would have been too time-consuming, the process is automated, selecting all functions identified by *rizin*, excluding imports, to be obfuscated.

The obfuscated programs had to be tested to ensure that they were working properly and had the expected output. The obfuscator had an overall success rate of 86%. Programs compiled with *tinycc* had the worst success rate of all, with only 25% of the executables leading to a properly working obfuscated file, followed by the programs previously obfuscated with tigress' Flatten and Virtualize transformations, with 87% and 81% success rate respectively. The programs compiled with the other methods worked as expected over 90% of the time. A breakdown of the working executables for each compiler is presented at table 6.2.

Further investigation revealed that *lief*, the library used to parse and edit the ELF was not able to correctly parse and edit some programs compiled with *tinycc*, leading to malformed executables that could not be executed.

| Compilation | Obfuscated |
|---|---|
| **All** | 1579.35% |
| **All (No tinycc)** | 168.77% |
| **Clang** | 182.07% |
| **compcertcc** | 170.38% |
| **gcc** | 164.26% |
| **tinycc** | 109447.51% |
| **Flatten** | 173.62% |
| **Virtualize** | 144.18% |

Table 6.3: Size overhead of obfuscated programs

## 6.2 Size

The size increase of the files created by the obfuscator over their original counterparts is shown at Table 6.3. Files created with the obfuscator had on average a size increase of 1579%. The results are inflated by the files obfuscated with *tinycc*, which had, on average, an overhead of over 109447%. Without counting them, the obfuscation method adds a file size overhead of around 168%.

## 6.3 Execution Time

The overhead introduced by the obfuscation is calculated by comparing the execution times of the original and obfuscated programs.

The hyperfine [36] benchmarking tool is used to calculate the execution time of the programs. Hyperfine allows to benchmark command line programs, automatically calculating the optimal number of times to execute each program. All programs were tested with the default configuration and 5 warm-up rounds.

Having $T$ as the original execution time of the program and $Tobf$ as the time of the obfuscated sample, the overhead can be defined with the following formula:

$$Overhead = T/Tobf$$

Table 6.4 shows the average overhead added by the obfuscation for each configuration. The overhead varies

| | Conf. 1 | Conf. 2 | Conf. 3 | Conf. 4 | Conf. 5 | Conf. 6 | Conf. 7 |
|---|---|---|---|---|---|---|---|
| **All (No tigress)** | 64.40 | 40.47 | 55.03 | 54.10 | 8.24 | 2.29 | 1.84 |
| **Clang** | 54.05 | 36.99 | 45.58 | 43.21 | 5.16 | 2.35 | 2.07 |
| **compcertcc** | 101.37 | 50.55 | 76.18 | 88.35 | 3.73 | 2.21 | 1.60 |
| **gcc** | 76.69 | 44.00 | 61.40 | 55.28 | 16.07 | 2.85 | 1.72 |
| **tinycc** | 261.53 | 159.38 | 210.34 | 246.08 | 134.76 | 8.44 | 1.15 |
| **Flatten** | 147.32 | 97.53 | 134.07 | 121.96 | 37.38 | 4.84 | 1.81 |

Table 6.4: Results of the execution time overhead tests.

greatly between compilation methods, showing why a large and varied sample is important when testing obfuscation methods. The files created with Tigress' virtualization transformation are not shown since the obfuscator was not able to properly identify the relevant functions and showed almost no variation over the original execution time.

### 6.3.1 Static Analysis

To test the strength against static analysis, the test programs are analysed with *rizin* [9], performing two different tests. The first compares the instructions of the main function of the original program and the main function of the obfuscated program, calculating the percentage of the instructions that are no longer in the program. The second looks at the control flow graph recovered by *rizin* and calculates the percentage of the basic blocks of the main function in the obfuscated program it was able to identify. This second method only looks at the positions and length of the basic blocks and not at the resulting
The objective of the static analysis tests is to both see how the obfuscation affected the original code, and what a disassembler can recover from the obfuscated code.
Table 6.5 presents the mean percentage of instructions from the original program that remain concealed following obfuscation with varying configurations. Meanwhile, Table 6.6 provides insights into the proportion of un-recovered basic blocks on the CFGs.

### 6.3.2 Dynamic Analysis

To test the resistance against automated dynamic analysis attacks, the obfuscated programs were tested against the *CFGEmulated* analysis from *angr*. This analysis from *angr* performs dynamic analysis by doing symbolic execution to extract the CFG of the program.[38]

| | **Conf. 1** | **Conf. 2** | **Conf. 3** | **Conf. 4** | **Conf. 5** | **Conf. 6** | **Conf. 7** |
|---|---|---|---|---|---|---|---|
| **All (No Tigress)** | 95.30% | 85.14% | 85.82% | 88.45% | 67.15% | 25.53% | 4.89% |
| **clang** | 95.31% | 85.14% | 85.83% | 88.45% | 67.16% | 25.54% | 8.04% |
| **compcertcc** | 97.19% | 89.42% | 90.64% | 95.30% | 61.54% | 9.45% | 1.00% |
| **gcc** | 94.02% | 88.66% | 87.82% | 91.87% | 50.79% | 15.65% | 2.40% |
| **tinycc** | 99.79% | 91.65% | 90.93% | 98.16% | 89.90% | 51.52% | 7.00% |

Table 6.5: Instructions hidden from static analysis tools.

| | **Conf. 1** | **Conf. 2** | **Conf. 3** | **Conf. 4** | **Conf. 5** | **Conf. 6** | **Conf. 7** |
|---|---|---|---|---|---|---|---|
| **All (No Tigress)** | 96.11% | 94.99% | 93.12% | 92.29% | 42.72% | 11.51% | 1.56% |
| **clang** | 97.93% | 95.48% | 96.36% | 96.76% | 62.18% | 12.49% | 1.41% |
| **compcertcc** | 97.95% | 100.00% | 100.00% | 97.95% | 3.06% | 0.39% | 0.00% |
| **gcc** | 94.22% | 93.39% | 89.05% | 87.51% | 32.82% | 13.39% | 2.06% |
| **tinycc** | 100.00% | 100.00% | 100.00% | 100.00% | 94.41% | 2.43% | 0.00% |

Table 6.6: Blocks hidden from static analysis tools.

| | **Conf. 1** | **Conf. 2** | **Conf. 3** | **Conf. 4** | **Conf. 5** | **Conf. 6** | **Conf. 7** |
|---|---|---|---|---|---|---|---|
| **Nodes** | 6.31 | 2.05 | 2.82 | 5.00 | 3.10 | 1.76 | 1.17 |
| **Links** | 11.17 | 3.21 | 4.64 | 8.74 | 5.12 | 2.62 | 1.58 |
| **Depth** | 0.69 | 0.55 | 0.58 | 0.65 | 0.68 | 0.63 | 0.56 |

Table 6.7: Results of the dynamic analysis tests with *angr*.

The test consists of loading the programs into an *angr* project with the *selfmodifying_code* flag set, which tells *angr* to take into account that code changes during execution.
The number of nodes and edges of the graph along with its maximum depth are recorded for all original programs and obfuscated programs to compare the results given by *angr* in both.
Table 6.7 contains the results of the test, showing the average variation of Nodes, Links and Maximum Depth between the original executables and their obfuscated versions created with each one of the configurations.

## 6.4 Results

This section discusses the metrics of using the data from the tests performed in the previous section. subsection 6.3.1 is used for both potency and resilience, subsection 6.3.2 is used to evaluate the dynamic analysis part of the resilience. The results in section 6.3 are used to evaluate the costs of the program. The stealth is discussed theoretically.

### 6.4.1 Potency

The potency of the obfuscator is measured by its strength against manual analysis. This makes the potency difficult to assess objectively, since different analysts will follow a different procedure when analysing the program. This section discusses the possible implications the obfuscation method has on manual analysis.
The code present in the obfuscated functions is not the original code that is executed, but fake code that will be replaced when a certain block needs to be executed. This acts as a first barrier to understanding the program. The results in Table 6.5 and Table 6.6 show that, with the default configuration, less than 5% of the original instructions are present in the obfuscated function. *Rizin* accepted the fake instructions placed by the obfuscator, generating the control flow graph using this data and reporting incorrect information based on it, identifying correctly only 4% of the blocks. This number goes even lower depending on the used compiler, with *compcertcc* and *clang* managing to hide almost 98% of the CFG of the programs.

The junk bytes also hide the call instructions to the transformation function, which makes it harder for the analyst to find where the program calls the deobfuscation function if he detects that the program is obfuscated.

Some of the calls included in the program are fake, and trying to deobfuscate a block by reverse engineering how the function of the obfuscation works might still lead to incorrect results, recovering the fake block instead of a real one. Furthermore, even if an analyst manages to identify all the calls to the transformation function he will not be able to know which ones are real and which ones are fake. To identify if a block is real or not, the analyst would first need to find a valid execution path that leads to that specific block. These blocks cannot easily be identified without running the program, since, because of the halting problem, it is hard to know if a program will arrive at a certain instruction, and therefore, if a block will ever be deobfuscated and executed.

A weak point in the potency of the obfuscator is that, in the current PoC, the code that hides and restores the code is not obfuscated, which could lead to an easier time understanding how the code is obfuscated. However, because of the properties discussed earlier in this section, this is not enough to deobfuscate the program, since the attacker would still need to find the real blocks.

### 6.4.2 Resilience

The resilience is measured by evaluating the ability of the obfuscation method to withstand automated attacks and tools. The results from subsection 6.3.1 and subsection 6.3.2 are used for this metric.

Against static analysis, the program managed to hide around 95% of the original instructions on average for the default configuration. This is due to blocks smaller than 5 bytes that are skipped by the obfuscator since they are smaller than the required size to use a call instruction. *Rizin* displayed the fake instructions introduced by the obfuscator as the instructions of the function and created erroneous CFGs using the information in the fake instructions. The configurations that limit the number of bytes that can be obfuscated for each block were not affected that much by the decreased number of bytes, with some cases even getting better results in the amount of blocks hidden. This might be *rizin* misinterpreting the incomplete instructions left unobfuscated at the end of blocks. On the other hand, limiting the obfuscated blocks to those larger than a minimum block size had a big impact on the results, reducing the obfuscated instructions to 67% on *Conf. 4* and as low as 4.89% for the *Conf. 7*.

*Angr*'s *CFGEmulated* did not manage to properly recover the CFG of the program. The generated CFG for programs generated with the default configuration had 6 times more nodes on average, and 11 times more links, while having less depth. The rest of the configurations show the number of nodes and links gets closer to the original values the fewer bytes we obfuscate every block or the larger the size requirement to obfuscate blocks is. The increased complexity of the returned CFG and the reduced depth indicates that the analysis was not able to properly recover the CFG while adding to the necessary effort that would be required by part of the analyst to interpret the results after running the tool.

### 6.4.3 Costs

The cost of the obfuscation method can be measured by the overhead in execution time and size observed in the obfuscated programs over the originals.

Table 6.4 shows the results of the runtime overhead tests, including the effect of the different configurations on the execution time. On average, programs where all blocks were completely obfuscated have a $x64$ increase in execution time over their unobfuscated counterparts.

The results of configurations 2 to 4 show that obfuscating only the first 8 bytes of each block reduces the execution time overhead by around 40%. Increasing the number of obfuscated bytes to 16 reduces the gain to only 15%, and further increases do not affect the cost of the obfuscation any discernible amount.

Obfuscating only blocks bigger than 16 bytes reduces the overhead by almost 90%. Increasing the minimum size of the blocks further to 32 bytes reduces the execution to barely two times the original.

These measures indicate that the smaller basic blocks are having the greatest effect on the execution time of the obfuscated programs, since to execute a small portion of code, the program has to run the function to deobfuscate it each time. Avoiding small blocks can help improve the cost of the method in exchange for worse *potency* and *resilience*, since changing these parameters also decreases the percentage of obfuscated code and the efficacy of the obfuscation against automated tools.

For the size of the obfuscated executables, Table 6.3 shows how the obfuscated programs have a size overhead of 168.77% on average.

The cost of the obfuscation method is overall relatively high, especially the cost it has on the execution time of the program. This limits the usage of this method on programs where efficiency is not a concern and for critical algorithms where reverse engineering holds a big risk.

### 6.4.4 Stealth

The stealth metric refers to the difficulty of detecting that a program has been obfuscated.

The obfuscator makes an effort to hide the calls to the obfuscation function by adding the initial bytes of *MOV* instructions at the end of the blocks that trick the disassembler into displaying the bytes from the

*CALL* instruction as part of the *MOV* instruction. This way only the calls at the start of functions or after a block that was not obfuscated will be shown in the disassembled code.

A drawback that reduces the stealth of this obfuscation method is that it requires to be able to write on the sections that contain the functions during execution, which is not the common behaviour on ELF executables and might be able to be used to identify that the program is obfuscated.

Lastly, the current PoC uses the same function to hide and restore code in each executable it creates, which could be used to create signatures to detect obfuscation using this method.

## 6.5 Comparison with previous similar studies

The obfuscation methods by Balachandran *et al.* [27] and Kanzaki *et al.* [29] work in a similar way to the proposed method. The first hides the jump instructions out of each block while the latter hides random instructions of the program.

The first study does not evaluate the resilience of the obfuscator but assesses the potency of the obfuscator by analysing the program with static and dynamic analysis tools. The program was disassembled with *IdaPRO*, a well-known reverse engineering program, which failed to properly identify an average of 80% of the instructions. The blocks of the control flow graphs of the program, on the other hand, only were hidden 60% of the time. The cost of this obfuscation method is significantly lower, this is to be expected since at most one byte is obfuscated per block, and each byte is restored by small blocks introduced in the function that each restore one block, and don't have to deal with identifying where the block that has to be obfuscated is located and which data corresponds to it, which would have a great impact when obfuscating small blocks that are going to be obfuscated multiple times on a single execution.

The second study does not provide any analysis of potency nor resilience, only discussing the theoretical properties of the obfuscation method, but provides an analysis of the performance overhead introduced in the program. The program gzip was obfuscated multiple times with different configurations, increasing the number of obfuscated instructions and comparing the execution times. The overhead increases linearly with the number of obfuscated instructions, ranging from 0% to 10% of the program, where the execution time goes from 0.25 seconds to 4.1 seconds, showing an overhead of execution time of 16.4 at the maximum number of obfuscated instructions. Apparently, the results show a slightly higher cost if compared to the results of configurations with a similar average of obfuscated instructions, like *Conf. 5* and *Conf. 6*. Further analysis should be done to properly compare the two obfuscation methods since these two configurations only obfuscate longer basic blocks, which might affect execution time.

The method proposed by Xiao *et al.* [23] uses a similar idea but uses it on top of virtualization, where the obfuscator hides the VM instructions of the basic blocks of the function instead of the actual instructions of the program, and encrypts them using AES keys embedded in the program. The methods used to calculate the potency and resilience of the obfuscation method, named the *Strenght* of the obfuscation in the paper, greatly differ from the methods used in this study, therefore the two methods are compared using the concepts of how they work. The use of Virtualization under the code block obfuscation adds a level of protection missing in this obfuscator, moreover, the algorithm used in this paper to hide the blocks is more complex, which might increase the time and effort needed to understand how the program is reconstructing the blocks. However, this is reflected in the cost of the method, the use of a more computationally expensive algorithm on top of an already slow obfuscation method appears to dramatically increase the execution time overhead. The results of the test show an overhead of between *43x* and *616x*, yet, only two programs were used to test the overhead, which is not enough empirical evidence to say with certainty how the obfuscation method affects the execution time overhead of the programs.

# 7 Conclusion

The aim of this thesis was to explore the design, implementation, and evaluation of a new variation of *Just-in-Time* obfuscation, and compare it with other obfuscation methods presented in recent years.

The obfuscator works by changing the contents of the basic blocks of the program with dummy instructions that are replaced by the real code once they need to be executed and then replaced back with the dummy instructions after execution. This approach increases the bar for analysts to understand the program's inner workings, hiding the flow of the program and retrieving the original code. The study also investigated possible additional protections that could make this obfuscation method resilient against known attack vectors identified during the literature review in chapter 3, some of which were also implemented, like the addition of fake blocks that are never executed, or the addition of junk code to hide the calls to the deobfuscation routine.

The results show that the method is potent and resilient but in exchange for an elevated cost. However, depending on the requirements, the obfuscation method can be configured to change how it selects what to obfuscate, allowing it to improve the cost in exchange for reduced potency and resilience.

The obfuscation method was able to trick static analysis tools into trusting the fake instructions added by the obfuscator, showing incorrect instructions and being unable to properly recover the CFGs. Dynamic analysis also failed to correctly extract the CFG of the program, finding many more blocks and links than the ones found on the original programs and with less depth, indicating that the analysis was not able to recover the original graph, and also creating extra work to analysts that decide to use it to reverse engineer the program. This indicates a better potency and resilience compared to other JIT obfuscators of previous studies, which obfuscate a smaller part of the code, making them more susceptible to automated and manual reverse engineering. The cost is the main drawback of the obfuscator. The executables obfuscated with it suffer an increase of over 60 times the execution time on average. Because of this this method of obfuscation would only be recommended for sections of code where efficiency is not of concern, or for key algorithms where reverse engineering poses a great risk, like the key validation function of a program. Another option would be to reduce the number of obfuscated blocks. Doing so drastically reduces the cost in exchange for lower potency and resilience, as seen in the tests. Furthermore, the study also found that factors like the compilation methods used to create the programs have a big impact on the cost of the obfuscator, so it should be used with programs compiled with *clang* or *gcc*, which had on average much better results.

## 7.1 Future work

In the context of this thesis, the presented methodology exhibits potential for enhancement and refinement. One avenue for improvement involves adding fake blocks, that are executed prior to reinstating the original block and its subsequent execution. Alternatively, integration with another obfuscation technique to augment the possible number of execution paths it can take could help bolster the obfuscation's potency and resilience.

Additionally, the PoC could benefit from refinement by obfuscating the function responsible for concealing and reinstating the code. This procedural modification would elevate the complexity of reverse engineering the obfuscation methodology, simultaneously introducing variability in the function employed for each obfuscated file. Such a measure serves to heighten detection challenges.

In terms of testing procedures, further investigation is needed to investigate why the obfuscation technique performance varies between executable files generated by different compilers. This inquiry should encompass an examination of how various methods manifest diverse cost implications, and in some cases, the incompatibility with certain compilers.

Lastly, it would be interesting to perform the same tests with other obfuscation methods from current literature to be able to perform a direct comparison between them.

# List of Figures

4.1 Representation of the effects of the obfuscation on the assembly code of the program. . . . . 20

5.1 Overview of the implementation of the obfuscator. . . . . . . . . . . . . . . . . . . . . . . . . . 29

5.2 Structure of the data section. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

# List of Tables

6.1 Information about basic blocks for each compilation method. . . . . . . . . . . . . . . . . . . 38
6.2 Successfully obfuscated files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
6.3 Size overhead of obfuscated programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.4 Results of the execution time overhead tests. . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.5 Instructions hidden from static analysis tools. . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.6 Blocks hidden from static analysis tools. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.7 Results of the dynamic analysis tests with *angr*. . . . . . . . . . . . . . . . . . . . . . . . . 43

# List of Listings

1 Structure of the ELF header . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Structure of the Shdr . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3 Structure of the Phdr . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

4 Example call to the obfuscation routine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

5 Modified hidden call to the obfuscation routine . . . . . . . . . . . . . . . . . . . . . . . . 25

6 Function to add a new section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

7 Code used to find the old and new IDs and call the xor code. . . . . . . . . . . . . . . . . . 33

8 XOR function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

# Acronyms

| | |
|---|---|
| AES | Advanced Encryption Standard |
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| ASLR | address space layout randomization |
| CFG | Control Flow Graph |
| COTS | Commercial-Of-The-Shelf |
| CPU | Central Processing Unit |
| DES | Data Encryption Standard |
| DRY | Don’t Repeat Yourself |
| ELF | Executable and Linkable Format |
| GCC | GNU Compiler Collection |
| GNU | GNU’s Not Unix |
| GUI | Graphical User Interface |
| I/O | Input/Output |
| IDE | Integrated Development Environment |
| IR | Intermediate Language |
| IT | Information Technology |
| JIT | Just-In-Time |

| | |
|---|---|
| JSON | JavaScript Object Notation |
| LIEF | Library to Instrument Executable Formats |
| OS | Operating System |
| PE | Portable Executable |
| Phdr | program header |
| PIE | position-independent executable |
| PoC | Proof of Concept |
| Shdr | section header |
| SLR | systematic literature review |
| VM | Virtual Machine |

# Bibliography

[1] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl, “Protecting software through obfuscation: Can it keep pace with progress in code analysis?” *ACM Comput. Surv.*, vol. 49, no. 1, Apr. 2016, ISSN: 0360-0300. DOI: `10.1145/2886012`. [Online]. Available: `https://doi.org/10.1145/2886012`.

[2] *Elf(5) linux user’s manual*, 1.12, Oct. 2003.

[3] Red Hat, *Position independent executables (pie)*, [Accessed 18-09-2023], Nov. 2012. [Online]. Available: `https://www.redhat.com/en/blog/position-independent-executables-pie`.

[4] Hector Marco-Gisbert and Ismael Ripoll Ripoll, “Address space layout randomization next generation,” *Applied Sciences*, vol. 9, no. 14, p. 2928, Jul. 2019, ISSN: 2076-3417. DOI: `10.3390/app9142928`. [Online]. Available: `http://dx.doi.org/10.3390/app9142928`.

[5] M. G. Rekoff, “On reverse engineering,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, no. 2, pp. 244–252, 1985. DOI: `10.1109/TSMC.1985.6313354`.

[6] Frances E. Allen, “Control flow analysis,” in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19, ISBN: 9781450373869. DOI: `10.1145/800028.808479`. [Online]. Available: `https://doi.org/10.1145/800028.808479`.

[7] Michael D Ernst, “Static and dynamic analysis: Synergy and duality,” in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[8] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann, “Demand-driven compositional symbolic execution,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3.

[9] Rizin Organization, *Rizin GitHub repository*, `https://github.com/rizinorg/rizin`, Dec. 2020.

[10] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, Jessie Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[11] Nick Stephens, Jessie Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.

[12] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[13] Bertrand Anckaert, Matias Madou, and Koen De Bosschere, "A model for self-modifying code," in *Information Hiding*, Jan L. Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–248, ISBN: 978-3-540-74124-4.

[14] Christian Collberg, Clark Thomborson, and Douglas Low, "A taxonomy of obfuscating transformations," *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, Jan. 1997.

[15] Rolf Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09, Montreal, Canada: USENIX Association, 2009, p. 1.

[16] Tihanyi Laszlo and Akos Kiss, "Obfuscating c++ programs via control flow flattening," 2009. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5061467`.

[17] Dongpeng Xu, Jiang Ming, and Dinghao Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Information Security*, Matt Bishop and Anderson C A Nascimento, Eds., Cham: Springer International Publishing, 2016, pp. 323–342, ISBN: 978-3-319-45871-7.

[18] Christian Collberg, *Jit*, [Accessed 18-09-2023]. [Online]. Available: `https://tigress.wtf/jitter.html`.

[19] Arini Balakrishnan and Chloe Schulze, "Code obfuscation literature survey," 2005. [Online]. Available: `https://api.semanticscholar.org/CorpusID:4152897`.

[20] Shi Dawei, Lv Delong, and Ye Zhibin, "Dynamic self-modifying code detection based on backward analysis," in *Proceedings of the 2018 10th International Conference on Computer and Automation Engineering*, ser. ICCAE 2018, Brisbane, Australia: Association for Computing Machinery, 2018,

pp. 199–204, ISBN: 9781450364102. DOI: `10.1145/3192975.3193016`. [Online]. Available: `https://doi.org/10.1145/3192975.3193016`.

[21] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, Eds., Cham: Springer International Publishing, 2018, pp. 372–392, ISBN: 978-3-319-93411-2.

[22] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl, "Sok: Automatic deobfuscation of virtualization-protected applications," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES 21, Vienna, Austria: Association for Computing Machinery, 2021, ISBN: 9781450390514. DOI: `10.1145/3465481.3465772`. [Online]. Available: `https://doi.org/10.1145/3465481.3465772`.

[23] Xuangan Xiao, Yizhuo Wang, Yikun Hu, and Dawu Gu, "Xvmp: An llvm-based code virtualization obfuscator," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 738–742. DOI: `10.1109/SANER56733.2023.00082`.

[24] Sebastian Schrittwieser and Stefan Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Information Hiding*, Tomáš Filler, Tomáš Pevný, Scott Craver, and Andrew Ker, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 270–284, ISBN: 978-3-642-24178-9.

[25] Andre Pawlowski, Moritz Contag, and Thorsten Holz, "Probfuscation: An obfuscation approach using probabilistic control flows," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, Eds., Cham: Springer International Publishing, 2016, pp. 165–185, ISBN: 978-3-319-40667-1.

[26] Shouki A. Ebad, Abdulbasit A. Darem, and Jemal H. Abawajy, "Measuring software obfuscation quality–a systematic literature review," *IEEE Access*, vol. 9, pp. 99 024–99 038, 2021. DOI: `10.1109/ACCESS.2021.3094517`.

[27] Vivek Balachandran and Sabu Emmanuel, "Potent and stealthy control flow obfuscation by stack based self-modifying code," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 4, pp. 669–681, 2013. DOI: `10.1109/TIFS.2013.2250964`.

[28] Gregory Morse, Midya Alqaradaghi, and Tamás Kozsik, "Control flow obfuscation with irreducible loops and self-modifying code," *Publicationes Mathematicae Debrecen*, 2022.

[29] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, “Exploiting self-modification mechanism for program protection,” in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, 2003, pp. 170–179. DOI: `10.1109/CMPSAC.2003.1245338`.

[30] John F. Reiser Markus F.X.J. Oberhumer László Molnár, *Ultimate packer for executables*, `https://upx.github.io/`, 2022.

[31] Xingwei Li, Zheng Shan, Fudong Liu, Yihang Chen, and Yifan Hou, “A consistently-executing graph-based approach for malware packer identification,” *IEEE Access*, vol. 7, pp. 51 620–51 629, 2019. DOI: `10.1109/ACCESS.2019.2910268`.

[32] Amit Nizri, *Unpacking upx*, Website, Acessed: Aug 2023, Apr. 2022. [Online]. Available: `%5Curl%7Bhttps://amitniz.github.io/posts/unpacking_upx/%7D`.

[33] Alessandro Strino, *Manually unpacking of packed executable*, Website, Acessed: Aug 2023, Sep. 2021. [Online]. Available: `%5Curl%7Bhttps://viuleeenz.github.io/posts/2021/09/manually-unpacking-of-packed-executable/%7D`.

[34] Sebastien Damaye, *Manually unpacking winupack*, Website, Acessed: Aug 2023, Mar. 2018. [Online]. Available: `%5Curl%7Bhttps://www.aldeid.com/wiki/Category:Digital-Forensics/Computer-Forensics/Anti-Reverse-Engineering/Packers/WinUpack%7D`.

[35] Romain Thomas, *Lief - library to instrument executable formats*, https://lief.quarkslab.com/, Apr. 2017.

[36] David Peter, *hyperfine*, version 1.16.1, Mar. 2023. [Online]. Available: `%5Curl%7Bhttps://github.com/sharkdp/hyperfine%7D`.

[37] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra, “Distributed application tamper detection via continuous software updates,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC ’12, Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 319–328, ISBN: 9781450313124. DOI: `10.1145/2420950.2420997`. [Online]. Available: `https://doi.org/10.1145/2420950.2420997`.

[38] *Control-flow graph recovery - angr documentation*, `https://docs.angr.io/en/latest/analyses/cfg.html`, [Accessed 06-09-2023].