CentraleSupélec

Internship Report

Automatic deobfuscation of virtualized binaries with LLVM

Désobfuscation automatique de binaires virtualisés avec LLVM

Reverse-Engineering; Deobfuscation; Virtualization-Obfuscation

Author:
Jack Royer

Supervisors:
Frédéric Tronel, Yaëlle Vinçont:
SUSHI, IRISA

Eléonore, Valentino:
Thales

Abstract: This report presents Mermaid: a new tool designed for automatic deobfuscation of virtualized binaries using LLVM. We use this tool to partially deobfuscate binaries obfuscated with the Tigress obfuscator. We provide a comprehensive introduction to the fields of binary analysis and deobfuscation before discussing the design, strengths and limitations of Mermaid. Although still very immature, our work provides evidence that automatic deobfuscation of large real world binaries is a reasonable possibility.

Contents

Introduction 1

I State of the Art 2

1 Binary analysis 2
- 1.1 Preliminary analysis . . . 2
  - 1.1.1 Recovering instructions . . . 2
  - 1.1.2 Intermediate Representation . . . 3
  - 1.1.3 Constructing a CFG . . . 3
  - 1.1.4 Recovering function boundaries . . . 4
- 1.2 Taint analysis . . . 4
- 1.3 Abstract interpretation . . . 4
- 1.4 Symbolic execution . . . 4
  - 1.4.1 Weaknesses and the path explosion problem . . . 4
  - 1.4.2 Concolic execution . . . 5
  - 1.4.3 SMT solvers . . . 5
- 1.5 Challenges of studying binary analysis . . . 5

2 Obfuscation 6
- 2.1 Evaluating obfuscation . . . 6
- 2.2 Threat modeling . . . 6
- 2.3 Simple code transformation . . . 6
- 2.4 Mixed Boolean Arithmetic . . . 7
  - 2.4.1 Chaining MBA . . . 7
- 2.5 Control flow flattening . . . 8
- 2.6 Code tamper proofing . . . 8
- 2.7 Virtualization . . . 8
  - 2.7.1 Hardening VM . . . 9
  - 2.7.2 Performance . . . 9
- 2.8 Summary . . . 10

3 Deobfuscation 10
- 3.1 Defining simplicity . . . 10
- 3.2 Objectives and constraints . . . 10
- 3.3 Compiler optimizations and LLVM . . . 11
- 3.4 CFG deflattening . . . 11
- 3.5 Defeating MBA . . . 11
  - 3.5.1 Pattern matching . . . 12
  - 3.5.2 SSPAM . . . 12
- 3.6 Defeating virtualization . . . 12

3.6.1 Abstract interpretation and VPC . . . 12
3.6.2 Pattern matching . . . 12
3.6.3 Program synthesis . . . 12
3.7 Taint analysis based strategy . . . 13
3.8 Triton . . . 13

II Internship work 14

4 Project scope 14

5 An introduction to LLVM IR 15
5.1 The design of LLVM IR . . . 15
5.1.1 Abstraction . . . 15
5.1.2 Example IR . . . 16
5.1.3 Static single assignment . . . 16
5.1.4 Phi nodes . . . 16
5.1.5 Memory in LLVM . . . 17
5.2 The often misunderstood GEP instruction . . . 18
5.2.1 Structures . . . 20

6 Initial analysis 22
6.1 Manual deobfuscation . . . 22
6.2 Automatic deobfuscation strategy . . . 25
6.3 Limits of Triton for automatic deobfuscation . . . 25
6.3.1 A hypothesis: starting with optimizations . . . 25

7 Mermaid: an automatic deobfuscation tool using LLVM 26
7.1 Deobfuscation strategy . . . 26
7.2 The obfuscated program . . . 26
7.3 Building a CFG . . . 27
7.3.1 Challenges with static analysis . . . 27
7.3.2 The motivation behind partial CFG . . . 27
7.3.3 Dynamic partial CFG construction . . . 27
7.3.4 Results . . . 28
7.3.5 LibCFG . . . 29
7.3.6 Typing a CFG . . . 30
7.4 Lifting to LLVM . . . 30
7.4.1 Advantages of code lifting . . . 31
7.4.2 Over-expressiveness . . . 31
7.4.3 Remill . . . 31
7.4.4 Remill intrinsics . . . 32
7.4.5 Saturn . . . 33
7.4.6 Recreating a CFG . . . 33
7.4.7 Wrappers . . . 35
7.4.8 Result . . . 36
7.4.9 Graph Edit Distance . . . 37
7.5 Taint analysis . . . 40
7.5.1 The interpreter . . . 40
7.5.2 Taint precision . . . 40
7.5.3 Limitations . . . 40
7.5.4 Results . . . 40

7.6 Reconstructing a CFG . . . 42
7.6.1 Strategy . . . 42
7.6.2 Implementation . . . 45
7.7 From CFG to LLVM module . . . 46
7.7.1 The challenge with SSA . . . 46
7.7.2 The "algorithm" . . . 47
7.7.3 Results . . . 48

8 LLVM optimizations **49**

9 Results **49**
9.1 Loop invariants . . . 49
9.2 After a nudge . . . 50
9.3 Going back to our initial program . . . 50

Conclusion **52**

Appendix **I**

A CFGs **I**
A.1 x86 CFG of the Collatz function . . . I
A.2 Partial x86 CFG of the obfuscated Collatz function . . . II
A.3 Partial lifted CFG of the obfuscated Collatz function . . . III
A.4 Reconstructed CFG of the `challenge01` program deobfuscated with Mermaid . . . IV

B Code **V**
B.1 Decompiled code of deobfuscated challenge01 function . . . V

References **IX**

Acronyms **X**

Acknowledgements

I would like to express my heartfelt gratitude to all those who have supported and guided me throughout the completion of this master's thesis. First and foremost, I extend my sincere thanks to all four of my advisors[1]: Eléonore, Frédéric, Valentino and Yaëlle, for their invaluable guidance, insightful feedback, and unwavering support. Their expertise and encouragement have been instrumental in shaping my internship. I am also grateful to the other members of Thales for their technical insight and camaraderie making this internship more enjoyable and enriching.[2]

During this internship I had the opportunity to discuss with external researchers who provided me with assistance. I would particularly like to thank the authors of SATURN: Peter and Matteo for their time and advice.

This internship concludes my master's degree, I would like to thank my family, friends and Rayray for their continuous support, understanding, and encouragement throughout this journey.

Finally, I acknowledge the financial support provided by Thales, which made this research possible. Their generosity has been crucial in enabling me to pursue this internship.

[1] By alphabetic order.

[2] And making me a better rock climber.

Introduction

The International Obfuscated C Code Contest was created forty years ago. The creators were inspired after reviewing code so intentionally convoluted that it transcended mere bad programming: *"It's more than bad code, the author really had to try to make it this bad!"* [30]. In the last forty years, software obfuscation has grown to have a pivotal role in copyright enforcement and cybersecurity.

Compiled programming languages, notably the C language, are compiled to machine code before being executed. This machine code is not easily understandable by humans. For a long time, this complexity was considered a sufficient protection from prying eyes.

However, the rise of disassemblers and binary analysis tools has enabled efficient analysis of binary files. Machine code was no longer enough to prevent analysis, and better protective measures were required.

To protect binaries from analysis, many strategies exist including hardware solutions, anti-debugging strategies and software obfuscation. This report focuses on obfuscation, which is the act of making a program harder to understand without altering its behavior. It is often used to enforce copyright or hide malicious intent.

Understanding and analyzing obfuscated programs is hard, by design. Dedicated tools are required for such analysis: deobfuscators. Deobfuscation is required to understand obfuscated code before adding functionality or interoperability with it. Another important use of deobfuscators is the analysis of obfuscated malicious software.

In cybersecurity, more precisely malware analysis, obfuscation plays a crucial role as many malwares are obfuscated. Malicious actors use obfuscation not only to complicate human analysis but also to bypass automatic analysis and detection. The widely recognized MITRE ATT&CK framework acknowledged the importance of obfuscation with a dedicated section for *Obfuscated Files or Information* [12].

Malware obfuscation can range from extremely simple techniques, such as the use of RC4 or base64 to obfuscate strings [22], to more sophisticated methods such as the use of the virtualization obfuscator VMProtect in the malware BackdoorDiplomacy [7] and KillDisk[20].

When complex obfuscators are used, specialized deobfuscators are required to understand the malware's behavior. During this internship we will focus on deobfuscating virtualized binaries. Although various strategies exist for automatic deobfuscation of virtualized binaries, we are curious if such strategies can be used on real world obfuscated binaries. We will therefore be focusing on scalability and performances. We will attempt to construct a prototype using state of the art deobfuscation strategies to demonstrate that significant performance gains can be made by leveraging LLVM.

Before constructing such a tool, we will introduce the field of binary analysis and deobfuscation. Beginning with an exploration of general binary analysis strategies and limitations, we then take a look at binary obfuscation and highlight the links with binary analysis weaknesses. Finally, we will examine deobfuscation, to understand what strategies are used and assess their effectiveness.

Following the state of the art, we will discuss the scope of our tool, and provide a brief introduction to LLVM. We will discuss our experiment with manual deobfuscation and then provide an in-depth presentation of the tool, showcasing the design decisions, results and limitations.

Part I
State of the Art

1 Binary analysis

Binary analysis seeks to extract semantics from an executable binary file. The Rice Theorem [40], a fundamental result in the theory of computability, states that: *"All non-trivial, semantic properties of programs are undecidable"*. Unfortunately, in the case of binary analysis, the Rice Theorem prevents the existence of any universal algorithm. Instead, binary analysis tools have to rely on heuristics that either lack complete coverage or exhibit occasional inaccuracies, creating a *coverage vs correctness trade-off*. Understanding this trade-off is important to better understand the strategies used in binary analysis.

1.1 Preliminary analysis

1.1.1 Recovering instructions

An important first step in binary analysis consists in recovering the instructions. As machine code is hard to understand and work with, binary analysis tools first try to *disassemble* it: that is, recover the assembly code of the program. Figure 1 illustrates the different shapes of a program: figure 1a contains source code, figure 1c contains the AMD64 assembly instructions and figure 1d contains the hexdump of a complete ELF binary. Although it is easy to decode single instructions, challenges arise when attempting to disassemble an entire program. The wide plurality of Instruction Set Architectures (ISAs) further complicate disassembly. For example, if we consider the widely used Intel architecture, specific challenges arise from the use variable width encoding and the mix of data in code, which this architecture allows. Wartel *et al.* showed that distinguishing data from code was an undecidable problem in *x86* binaries, as this problem can be reduced to the famous halting problem [55]. Because of this limitation, heuristics have to be used, which do not guarantee correctness but provide an approximation. Different strategies can be used such as *linear sweep* for better coverage or *recursive descent* for better correctness [36].

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

(a) Original C program

```
@.hello = constant [14 x i8] c"Hello, World!\00"
declare i32 @printf(ptr, ...)

define i32 @main() {
  %1 = call i32 @printf(ptr @.hello)
  ret i32 0
}
```

(b) Corresponding LLVM IR program

```
main:
    push    rax
    lea     rdi, [rip + .L.str]
    xor     eax, eax
    call    printf@PLT
    xor     eax, eax
    pop     rcx
    ret

.L.str:
    .asciz  "Hello, World!"
```

(c) Corresponding Assembly program

```
7f454c460201010031c0ffc089c7eb1802003e000100000008
800200000000003a00000000000000be6a80020031d2b20eeb
1f0040003800010001000000050000000000000000000000000
800200000000000f05b03c31ff0f0578000000000000007800
0000000000048656c6c6f2c20776f726c64210a
```

(d) Nathan Otterness' "Hello, World!" ELF binary [35]

Figure 1: Different representations of a "Hello, World!" program

1.1.2 Intermediate Representation

Intermediate Representation (IR) is an optional step that facilitates further analysis. Many binary analysis tools [38] [32] [17] (as well as obfuscation [46] and deobfuscation tools [18]) rely on IRs. An IR is a new language on which analysis is easier to perform. This new language often offers more abstractions. For example, LLVM's IR [29] allows the use of types, variables and calls with arguments. Figure 1b contains the LLVM IR equivalent of the C code in figure 1a. Lifting binaries to an IR can be a bottleneck for binary analysis tools, so this step should not be taken lightly [24]. IRs also allow the reuse of underlying logic as many ISAs can be lifted to the same IR onto which the analysis is performed [16]. IRs are therefore fundamental for writing cross-platform tools.

1.1.3 Constructing a Control Flow Graph

One of the fundamental structures constructed by a binary analysis tool is the Control Flow Graph (CFG). A CFG is a representation of a program's control flow, often inside a function. The program is separated into *basic blocks* often connected to each other with conditional jumps. In a basic block, all the instructions are guaranteed to be executed sequentially. Figure 2 depicts a CFG for a function calculating the *Collatz flight time*.

Constructing a CFG can be tricky. For example when building a CFG, one needs to identify jump instructions and jump targets, this information is particularly hard to retrieve in the case of indirect jumps. Dennis Andriesse *et al.* found that the accuracy of a constructed CFG was closely linked to how well the instructions were recovered [1]. Many complex strategies exist to construct a CFG such as the use of symbolic execution [48]. Other tools such as Radare2 [38] rely on a complex set of handcrafted heuristics [36]. An important aspect of a CFG is its shape. An analyst can easily distinguish different control flows (`if / else`, `while`, `switch`) simply based on a CFG's shape. Some of these shapes are illustrated in Figure 3.

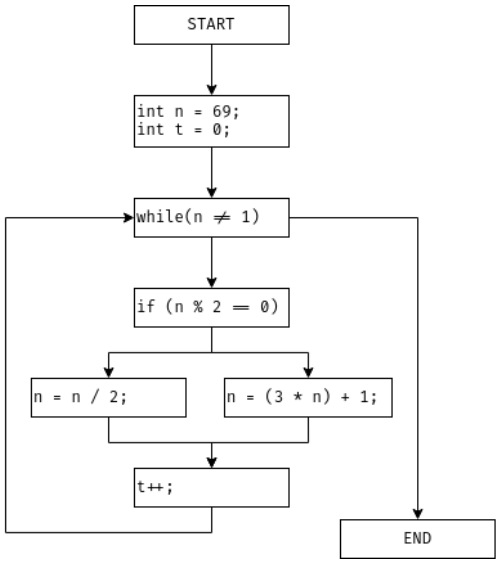
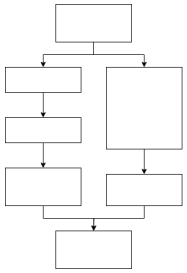
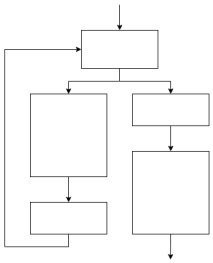


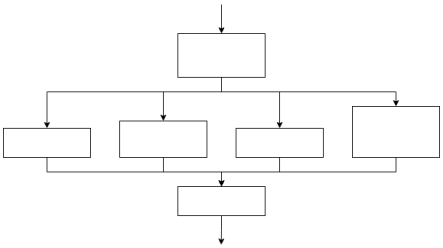
Figure 2: A CFG for a function calculating the Collatz flight time



(a) If / Else



(b) Loop



(c) Switch

Figure 3: CFGs shapes

1.1.4 Recovering function boundaries

To facilitate analysis, isolating and independently analyzing functions is crucial. Although function locations might be present in a binary file (for example `eh_frame` in ELF files), it is not always the case (and even less so the case in an obfuscated binary). In a binary file where functions are not explicitly demarcated, finding their boundaries is challenging. Certain ISAs and IRs may not even have an explicit call instruction [16]. Although compilers often add recognizable prologues and epilogues to functions, these are not always present. Recovering function boundaries is especially challenging in the case of non-returning functions, which makes it difficult to separate two adjacent functions [36]. This problem also arises from compiler optimizations like function inlining (where the body of a function is pasted at each, or some call sites), or for tail-call optimization.

1.2 Taint analysis

Taint analysis is a strategy to analyze data flow within a program. It can be performed either statically or dynamically. During dynamic taint analysis, certain initial variables are *tainted*, and the taint is propagated during the program's execution to any variable that depends on tainted data. The resulting tainted data depend on the initial tainted ones, thus providing a clear picture of data dependency. This strategy is often used to determine what parts of the code depend on a user input, thereby assisting in vulnerability detection. Taint analysis is a frequently used both in binary analysis tools, such as Angr [48], and deobfuscation tools, such as Loki attack [46].

1.3 Abstract interpretation

As discussed earlier, most semantic properties of a program are not decidable. In 1977, Cousot and Cousot introduced abstract interpretation as a solution to this problem [13]. Abstract interpretation attempts to identify a tight over approximation of the program states. To do so, the program is lifted to an *abstract domain*. A simple example of an abstract domain is an interval-based numerical abstract domain. At every point of the program, an interval is assigned to each variable of the program state. The changes are then propagated until a fix-point is reached. In real binaries, more complex abstract domains are required. The binary analysis tool Angr [48] describes a modified version of the Value-Set Abstract domain to work on binaries [48]. Abstract interpretation can be particularly useful when performing certain optimizations such as dead code elimination, and constant propagation. Abstract interpretation is also commonly used in binary analysis tools, such as BINSEC [17] or BinCAT [5].

1.4 Symbolic execution

Symbolic execution is one of the most promising strategies for binary analysis [9]. It works by using *symbolic values* (symbols) in addition to *concrete data values*. The state of the program can now be expressed as a set of *symbolic expressions*. A symbolic execution engine then tries to explore as many execution paths as possible, adding constraints to symbols when required (for example in a conditional branch). Finally, these constraints are solved using a Satisfiability Modulo Theory (SMT) solver, providing insight into the different variables.

1.4.1 Weaknesses and the path explosion problem

Symbolic execution is not perfect and is often computationally expensive. On a branch, a symbolic execution engine *forks* and explores both sides. In other words, a set of constraints is created for each path. This is especially problematic when a lot of paths are explored such as in the case of loops. This explosion in computational cost is known as the *path explosion problem*. The symbolic execution engine can no longer run with decent performance or in the worst cases, the program runs out of memory. This is not the only limitation of symbolic execution engines. They can also be limited by SMT solvers not being able to solve certain equations. Furthermore, symbolic execution engines require access to code to perform their analysis

which is not always accessible (or very large), when programs interact with their environment (system calls). Symbolic execution engines also face difficulties when trying to model memory. The choice of memory model can greatly influence the coverage of a symbolic analyzer [9]. Some tools, such as CUTE [47], only allow equality operation on pointers whereas others, such as KLEE [8], model memory using the theory of arrays.

1.4.2 Concolic execution

The SOK article *Symbolic Execution for Software Testing: Three Decades later* covers a strategy solving some of these issues: *concolic execution* [9]. Concolic execution: literally *concrete symbolic execution*, is an alternative to symbolic execution where the program is also run concretely. This allows the engine to revert to concrete values if too many paths appear or if the program is interacting with its environment. Concolic execution sacrifices some completeness for better performances and better accuracy. Indeed, although only a single path might be explored (loss of completeness), it is guaranteed that this path is reachable as a concrete value reached it (increase of accuracy). Without such a concrete value, certain states might have incompatible constraints that the SMT solver did not pick up on. For this reason concolic execution is used in the binary analysis tool Angr [48] to find vulnerable states.

1.4.3 SMT solvers

The solvers used for symbolic execution are Satisfiability Modulo Theory (SMT) solvers. The most popular solver is Z3 [15]. Developed by Microsoft, Z3 is a very performant SMT solver which can also perform boolean and vector operations making it ideal for binary analysis. Although Z3 is the most popular option for historic reasons, today, it is not always the most performant solver. SMT Comp is an annual competition ranking SMT solvers. In 2023, SMT Comp found that Bitwuzla [34] was the best solver in the `QF_Equality+Bitvec` track, a track in which Z3 did not compete in [49]. It is important to use the proper solver for a given problem.

1.5 Challenges of studying binary analysis

Binary analysis is a hard problem, but the study of binary analysis is even harder. Academia is plagued with many issues when it comes to the study of binary analysis. The largest difficulty is establishing a ground truth. Indeed, many binary analysis programs are specialized for certain applications. The data set on which these programs are tested can introduce a bias in the results. Results vary significantly depending on the hypotheses and binaries analyzed. Furthermore, there is a significant issue with systematization and reliability which is added to this problem. Indeed, papers even with the same dataset can find different results (often due to version differences and/or the random nature of many analysis tools). Some papers even call out others for spreading wrong results: “a recent study (unintentionally) overstated the accuracy of linear sweep” [36].

2 Obfuscation

Obfuscation is the process of making a program harder to understand while preserving the original program's behavior. One would want obfuscation to create a perfect *virtual black box* making it impossible for an attacker to analyze it. However, in 2001, Barak *et al.* showed that such *perfect* obfuscation was impossible [3]. Instead, practical obfuscation aims to hinder (rather than prevent) analysis.

Obfuscation can be performed either on source code (for example: Tigress [10]), or as part of the compilation chain (for example: Obfuscator-LLVM [25]). Obfuscated binaries are often less efficient compared to the original non obfuscated binaries. Obfuscated binaries can be around 1000× larger than the original binary and run around 50× slower [46]. For that reason, it is often not recommended to obfuscate the entire program. By restricting obfuscation to small critical areas, the performance overhead becomes manageable. A malware author might obfuscate an encryption routine or communications with a command and control server.

The authors of *Loki* [46] have a very illustrative case study on the real world cost of obfuscation. They obfuscate a function that decrypts DVD keys from Libdvdcss. This function is run only once, and is a prime target for crackers. They found that obfuscating this function slowed its execution from 2,952 nanoseconds to 937,606 nanoseconds. This difference, although of multiple orders of magnitude, is negligible for the user and a justifiable cost to protect intellectual property [46].

2.1 Evaluating obfuscation

In their seminal paper on obfuscation, Collberg *et al.* [11] define four important metrics for obfuscation :

1. *Potency* measures complexity. A transformation is potent if it makes the program more *complex* (less understandable). The definition is by design quite vague, but the authors give examples of such transformations: increasing program size, increasing the amount of predicates, increasing the amount of function arguments.

2. *Resiliency* measures how well a transformation can resist automatic deobfuscation. While potency seeks to confuse human analyst, resiliency seeks to confuse binary analyzers.

3. *Stealth* measures how well obfuscated code *fits* into the original program. That is how well a human analyst would be able to spot the obfuscated code.

4. *Cost* refers to how these transformations will affect the program's performance and size.

2.2 Threat modeling

Before obfuscating a program, it is important to properly characterize the threat model and understand the capabilities of an attacker. Is the attacker interested in fully understanding the program's semantics, or are they seeking to carve out a specific piece of valuable information (e.g. a cryptographic key)? Will the attacker know pieces of logic used in the program? Will the attacker have access to the location of the obfuscated code they want to analyze? Will the attacker be able to perform dynamic analysis? [3]. These questions can determine what the best obfuscator to use is and what needs to be protected.

2.3 Simple code transformation

The simplest form of obfuscation relies on many simple code transformations that can be applied to a program to make it less intelligible. Although on their own they are not necessarily resilient, these transformations can be added after other strategies to further obfuscate a program.

[3] Dynamic analysis is not always possible, for example in embedded systems

- *Dead code* is code that cannot be reached during execution. Adding dead code will add surface the attacker has to analyze, and can often include red herrings. It is not uncommon for there to be significantly more dead code than actual code, for example by drowning a malicious executable into a well known library. Adding dead code and bloating binaries can also be obtained by statically linking binaries.
- *Irrelevant code* is similar to dead code in that it is additional code that does not interfere with the program's behavior. However, unlike dead code, irrelevant code is reached during the program's execution and simply does nothing.
- *Code cloning* is yet another strategy to add confusion to a program. A basic block with multiple entries can be *cloned* into multiple identical basic blocks with a single entry.
- *Opaque predicates* in their simplest form are conditional expressions used in branches that do not matter. These expressions can always evaluate to true (P^T), or to false (P^F). Another option is that the value of the predicate does not matter: both branches might perform the same code ($P^?$). In most cases, the opaque predicate is obfuscated itself to not be obvious.
- *Compiler optimizations* can also be used to obfuscate programs. Examples of such optimizations are loop splitting, loop reordering, function splitting, function reordering, overlapping code, overlapping instructions, *etc...*

Many obfuscators take advantage of existing compiler optimization passes to further obfuscate their code [46].

2.4 Mixed Boolean-Arithmetics

First introduced by Zhou *et al.* [58] in 2007, Mixed Boolean-Arithmetics (MBAs) expressions are expressions mixing arithmetic operators ($+$, $-$, $\times$) with boolean operators ($\neg$, $\oplus$, $\wedge$, $\vee$). When these expressions are mixed, it becomes very difficult to analyze them and find solutions. This difficulty stems in large part from the lack of general rules to interact between these operators (e.g. no distributivity, no associativity, ...). Obfuscators can introduce MBA expressions by replacing equivalent arithmetic expressions. For example:

$$(x \oplus y) + 2 \times (x \wedge y) = x + y$$

When provided with complex MBA expressions, SMT solvers are no longer able to simplify them or solve for given value [19]. This could confuse an analyst or block a symbolic execution engine. Furthermore, MBAs become even more confusing after going through compiler optimizations [19].

2.4.1 Chaining MBAs

MBAs are often created by replacing patterns with precomputed MBAs. This can make basic MBAs very vulnerable to pattern matching. A proposed solution is to chain MBAs. In a complex expression, subexpressions are first replaced with an MBA this process is iterated on the new expression. It is important to note that if the initial expression is not prioritized, then this transformation will mainly be modifying MBA elements rather than elements from the initial expression [46].

2.5 Control flow flattening

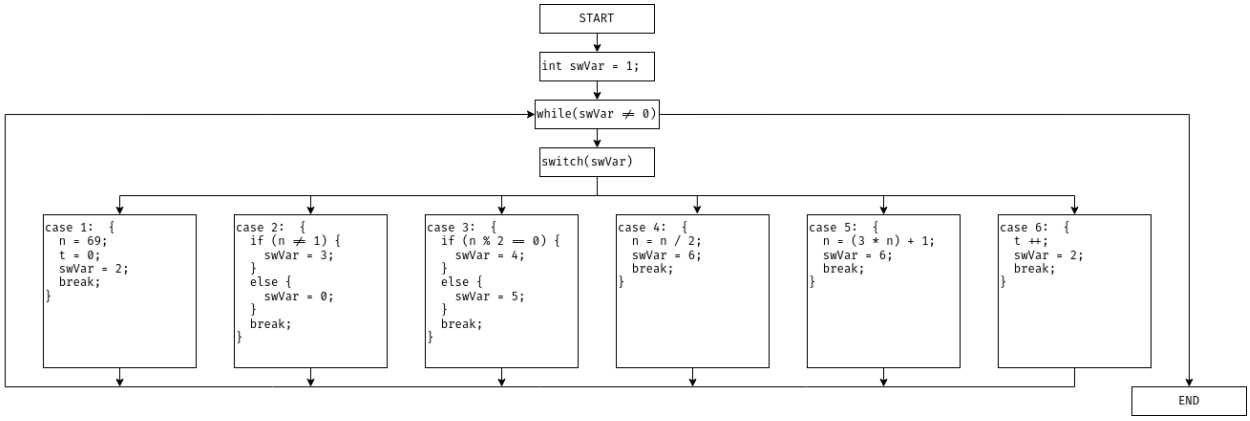


Figure 4: A CFG for a flattened Collatz flight time (*inspired by [28]*)

Control flow flattening is a strategy to make CFGs unintelligible. It is achieved by taking all the basic blocks that were originally nested and placing them next to one another. This structure is wrapped in a *dispatcher* (for example a switch statement) which is itself wrapped in a loop. The control flow inside our program is maintained by a variable representing the program's current state. Figure 4 is an example of a possible flattened CFG for the Collatz program. In the original CFG (Figure 2), the loop and branch were easily identifiable (Figure 3 illustrates distinctive CFG shapes). It is no longer the case in this flattened version. In Figure 4, a loop was used for the dispatcher and the variable `swVar` was used to represent the program's state. It is important to note that in a real obfuscated binary, many variables can be used to represent the program state.

Obfuscation through control flow flattening has a significant effect on a program's speed (up to 6× slower on toy examples) and size (up to 3× larger on toy examples) [28].

2.6 Code tamper proofing

During dynamic analysis, an attacker can often gain insight by modifying a binary. For example, by skipping a conditional branch, an attacker could obtain a trace on a sensitive part of the program. Code tamper proofing techniques are used to ensure that the machine code has not been modified. A popular technique used in oLLVM consist of calculating 32-bit CRCs over segments of machine code [25]. The results are used to modify the control flow in flattened code, altering the program's behavior if it has been tampered with. An attacker modifying the machine code, now also has to modify all the CRCs of segments which contain the modified code. This technique is on the verge of obfuscation and anti-debugging, but it is added to this report as an analyst working on obfuscated code is likely to be confronted with it. Furthermore, the code tamper proofing implementation described in this paragraph has an effect on the CFG, thus obfuscating even more the program.

2.7 Virtualization

Kochberger *et al.* describes virtualization as "one of the strongest techniques for obfuscating an application" [27]. Virtualization takes control flow flattening to an extreme. In a virtualized binary, the program is compiled to a made up instruction set. The binary contains the compiled bytecode as well as an interpreter. To make sense of the compiled bytecode, an analyst needs to reverse engineer the interpreter first. Virtualized binaries contain several distinctive components [52]:

1. **VM entry/exit**: performs the context switching between our Virtual Machine (VM) and its environment. It is comparable to a function's prologue or epilogue.

2. **VM dispatcher**: runs a fetch, decode, execute loop which runs our made up instructions. The VM is most vulnerable during the decode stage as that is what connects instructions to logic.

3. **Handler table**: contains the semantics of our custom instruction set.

These components can be seen in figure 5.

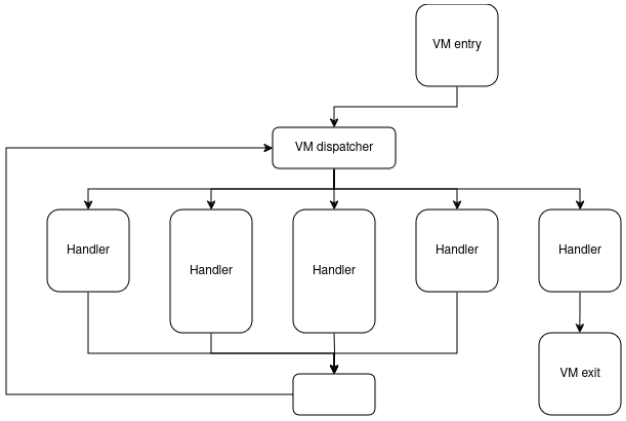


Figure 5: A schematized CFG of a virtualized program

2.7.1 Hardening Virtual Machines

To harden the VM and prevent simple pattern matching, virtualization obfuscators can construct an instruction set from a given binary, grouping logic into a single instruction. These complex, program specific instructions cannot be given a simple mnemonic. An analyst might struggle to make sense of such an instruction. Another strategy to harden instructions consists of merging core semantics [46]. By adding an argument to the instruction (the key), it is possible to select the desired semantic.

$$f(x, y, k) := e_0(k) * (x + y) + e_1(k) * (x - y) \tag{1}$$

$$e_i(i) := 1 \tag{2}$$

$$\forall k \neq i,\ e_i(k) := 0 \tag{3}$$

In the example, function f is a function defined in equation 1 and k is the key. This function satisfies $f(x, y, 0) = x + y$ and $f(x, y, 1) = x - y$.

2.7.2 Performance

The main challenge with virtual machines is their very high cost. This is understandable as the VM needs to lift the bytecode. In *Loki: Hardening Code Obfuscation Against Automated Attacks*, the authors found that cryptography primitives obfuscated with virtualization could be slowed by a factor of up to $15,000$ [46]. This performance toll is not negligible and is significantly greater than the other obfuscation strategies put forward in this document. This performance cost is what usually prevents VMs from being nested.

Obfuscations techniques and tools				
Strategy	Potency	Resiliency	Cost	Tools
Dead & Irrelevant code	+			oLLVM [25], Tigress [10],
MBAs	+++	+++	+	Loki [46], Tigress [10],
Tamper proofing		+++	+	oLLVM [25], Tigress [10],
Control flow flattening	++	++	++	oLLVM [25], Tigress [10],
Virtualization	+++	+++	+++	Loki [46], Tigress [10], VMProtect [54]

Table 1: Obfuscation techniques and tools (based on impressions in articles)

2.8 Summary

In addition to the obfuscation strategies seen in sections 2.3, 2.4, 2.5, 2.6 and 2.7, several other obfuscation strategies exist. These include, among others, *nanomites* from *Armadillo* [2] (or other similar techniques that rely on specific operating system features to transpose code logic) and encryption of strings or code sections. A summary of the obfuscation techniques described in this report can be found in Table 1. This table was created based on impressions in the articles rather than experimental data or prior works.

3 Deobfuscation

Deobfuscation is the act of making obfuscated software more understandable to humans and tools. It is made possible by the impossibility of creating perfect obfuscators. Deobfuscation is a game of cat and mouse with obfuscator authors. Certain deobfuscators are meant to be generic (VMAttack [53], VMHunt [56]) whereas others can be purposely built for given obfuscators. Certain researchers even go so far as to publish a deobfuscator alongside their obfuscators (Loki and Loki Attack [46]). Binary deobfuscation heavily relies on binary analysis tools.

3.1 Defining simplicity

There is a quite difficult concept to define when working on deobfuscation, which is simplicity. The aim of a deobfuscator is to revert the transformations done by an obfuscator, and as such a deobfuscator needs to make a program less complex: *simpler*. We already discussed (when defining *potency*) how defining anything linked to human understanding was complex. When defining *potency*, we got away with examples and a vague definition, but the problem remains.

Let's take the example of *simplifying* MBAs as a case study. In their article *Defeating MBA-based Obfuscation*, the authors [19] discuss the issue of simplicity noting that the most reduced form of an arithmetic expression is not necessarily the easiest one to understand. Sometimes, the factorized form of an expression is more readable such as in Equation 4, whereas other times, the expanded form is much better such as in Equation 5.

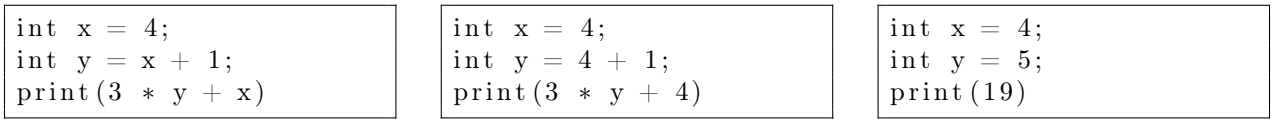
$$(1+x)^{100} = 1 + 100x + \cdots + 100x^{99} + x^{100} \tag{4}$$

$$(x-1)(x+1)(x-i)(x+i) = x^4 - 1 \tag{5}$$

The same issue arises when simplifying boolean expressions, and in general when attempting to simplify programs. Many deobfuscators and articles do not directly address how they define simplicity, and instead seem to only focus on retrieving the original program [27].

3.2 Objectives and constraints

Similarly to how obfuscation was concerned about attacker models, it is important to understand the constraints of a deobfuscator. An important consideration is whether the deobfuscator is meant to be automatic



(a) Example program (b) After *constant propagation* (c) After optimization passes

Figure 6: Compiler optimizations

or whether it relies on a human analyst. Other factors to consider are execution time and computational resources. Indeed, performance can be a significant bottleneck. In Kochberger *et al.*'s SOK article [27], deobfuscators are regularly described as crashing and running out of memory. Finally, any prior knowledge such as knowing the obfuscator, can significantly aid the deobfuscation process.

3.3 Compiler optimizations and LLVM

Compiler optimization is one of the most powerful strategies for simple deobfuscation. Two strategies stand out when it comes to reversing less resilient obfuscation strategies (for example dead code). Compiler optimizations heavily rely on *constant propagation* and *constant folding*.

- **Constant propagation**: Constant propagation is a compiler optimization strategy that "replaces" constants with their values in the program. In Figure 6a, we notice the variable `x` was replaced with its value: `4`.
- **Constant folding**: Constant folding is another compiler optimization strategy that computes expressions containing only known constants. In Figure 6b, we notice that the initial value for our variable `y` went from `4 + 1` to `5`.

By iterating these strategies, new constants are discovered (the variable `y` in Figure 6), then propagated until the program is fully simplified. Other simple optimizations can now be performed, such as removing unused variables.

An effective strategy to perform compiler optimizations is to use the same Intermediate Representation (IR) as a compiler. As we saw in section 1.1.2, using IRs is often useful for binary analysis, here is another advantage. Saturn [21] uses LLVM IR to deobfuscate binaries. They use their tool to reconstruct CFGs and solve opaque predicates using LLVM IR optimizations. The main challenge with this strategy is the need to lift the code to LLVM IR. To do so, Saturn uses Remill [39] as well as custom logic and passes.

3.4 CFG deflattening

At first, CFG deflattening might seem like a simple task: locate the dispatcher, then isolate and link the basic blocks back together. However, the main difficulty with deobfuscating control flow flattening stems from compiler optimizations which alter the program's structure [18]. Indeed, compiler optimizations can split and merge basic blocks. This makes recovering the original basic blocks as well as identifying the dispatcher quite difficult. CaDeCFF solves this challenge by focusing on the state variable instead of focusing on the program's structure [18]. CaDeCFF is able to find the state variable by counting how many branch conditions are influenced by each variable. Variables involved in many branch conditions are more likely to be the state variables.

3.5 Defeating MBA

SMT solvers are often unable to simplify or solve complex MBAs. In particular, although Z3 has a `simplify` function, its unable to simplify or find solutions to complex MBA expressions.

3.5.1 Pattern matching

Obfuscators often only use a handful of MBAs (TIGRESS only has 16 [46]). If these expressions can be recognized, they can be replaced with their equivalent known arithmetic or boolean expressions. Although this strategy is efficient, it does have its limitations in certain use cases. These limitations include combined MBAs, as well as not knowing which MBAs are used (for example a closed source or unknown obfuscator).

3.5.2 SSPAM

Eyrolles *et al.* offer a different method for defeating MBAs in their tool SSPAM [19]. The MBA expression is first transformed into a *term graph* (Figure 7). Transformations are then applied to the tree to reduce both the amount of nodes in the tree and the *MBA alternance*. MBA alternance refers to the amount of edges connecting arithmetic expressions to boolean expressions (these edges are depicted as dashed lines in Figure 7). In order to identify which transformations to apply, SSPAM mainly relies on pattern matching. However, it also uses Z3 to build alternative equivalent expressions in order to maximize its odds of identifying known patterns.

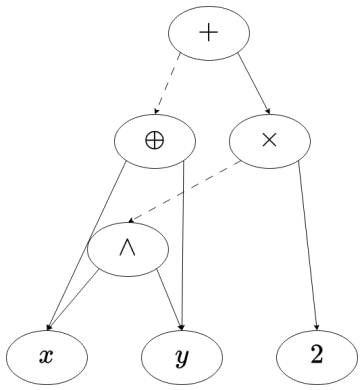


Figure 7: A term graph for the expression $(x \oplus y) + 2(x \wedge y)$

3.6 Defeating virtualization

Defeating virtualization based obfuscation is not a mundane task. In his article *Towards Static Analysis of Virtualization-Obfuscated Binaries*, Kinder shows that many traditional static analysis strategies (including abstract interpretation) are not able to tackle virtualization [26]. Several strategies exist, with varying levels of automation and results, this report only covers a few of those.

3.6.1 Abstract interpretation and Virtual Program Counters

Although "regular" abstract interpretation is not able to defeat virtualization, by adding another "dimension" to an interpreter, a more efficient VM deobfuscator can be built. Kinder found that by adding information about the Virtual Program Counter (VPC) to an abstract interpreter, he was able to extract a CFG from toy example virtualized binaries [26]. By *lifting* the abstract domain, abstract values for variables are no longer just computed at a program location. Instead, they are computed for each pair (program location, VPC).

3.6.2 Pattern matching

This strategy is especially popular with tools targeting a single obfuscator. These tools start by identifying handlers using their knowledge of the obfuscator. Then, once a handler has been identified, symbolic execution and taint analysis can be used to extract the semantic from the handler. For example, the deobfuscator LokiAttack, which was built to evaluate the obfuscator Loki, was built using this architecture [46]. If dynamic analysis is possible, the deobfuscator can also retrieve traces for the handler.

3.6.3 Program synthesis

Program synthesis is a different approach to dealing with virtualized binaries. It is especially powerful when confronted with obfuscated code having high syntactic complexity, yet low semantic complexity. In program synthesis, the handlers are treated like black boxes and semantics are extracted by observing inputs and outputs. By knowing expected operations as well as the input and output behavior of such operations, one is able to "synthesize" the handlers after observing their input/output behavior. Blazytko *et al.* pioneered program synthesis with their deobfuscator Synthia [6].

3.7 Taint analysis based strategy

Yadegari *et al.* [57], use a mix of heuristics and taint analysis to rebuild CFGs from virtualized binaries. They successfully test their stragtegy on multiple binaries and real world obfuscators including Themida [50], VM-Protect [54] and Tigress [10]. Although they do not reconstruct deobfuscated binary code, their approach and tests were a great inspiration for many parts of this internship.

3.8 Triton

The final strategy we will discuss is the one used by Triton [43]. Using symbolic execution, Salwan was able to automatically deobfuscate virtualized binaries including multiple binaries from the Tigress challenge.

The approach relies on dynamic taint analysis to identify which parts of the program are dependent on the inputs and outputs. The tainted data is then used to partially reconstruct the original CFG.

Using several traces, the deobfuscator reconstructs the original control flow. The inputs used to compute the traces are provided by a symbolic execution engine which attempts to maximize code coverage. The traces are then merged and the resulting program is simplified with LLVM.

Part II
Internship work

4 Project scope

During this internship, we focused on very simple functions with the following specifications:

1. The input is an Integer
2. The output is an integer
3. The function contains no calls

This model includes hash functions which are typically the kind of proprietary functions to be obfuscated, as discussed by Jonathan Salwan *et al.* [43].

The programs we worked on follow the template described in figure 8. The `main` function reads an input, calls the `SECRET` function, then displays the result.

```
void SECRET(unsigned long *input, unsigned long *outputs);

int main(int argc, char **argv) {
  unsigned long inputs[2] = {};
  unsigned long outputs[2] = {};

  if (argc != 2) {
    printf("Call this program with %i arguments\n", 1LL);
    exit(-1);
  }

  for (size_t i = 0; i <= 0; ++i) {
    unsigned long v5 = strtoul(argv[i + 1], 0LL, 10);
    inputs[i] = v5;
  }

  SECRET(inputs, outputs);

  for (size_t j = 0; j <= 0; ++j)
    printf("%lu\n", outputs[j]);

  return 0LL;
}
```

Figure 8: Program template

For our virtualization obfuscator, we chose Tigress [10]. This state of the art obfuscator allows users to pick which obfuscation to apply. In our case, we only want to apply virtualization.

The Tigress authors also published several challenges[51], following the template shown in figure 8. This template allows the obfuscated program to depend on a user input, all while preventing the need to handle calls and system calls in the obfuscated function.

5 An introduction to LLVM IR

LLVM was initially designed as a set of modular tools for building "great compilers". One of the many great features of LLVM is its intermediate representation - colloquially known as **LLVM IR** or simply **IR**.

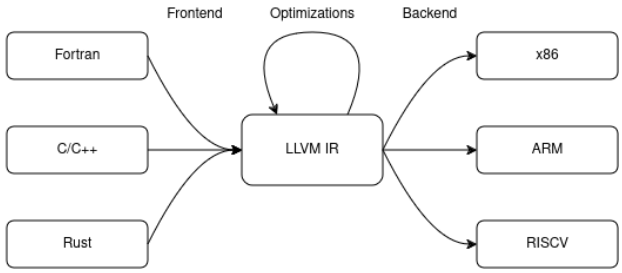


Figure 9: Schematized representation of LLVM's architecture

As seen in figure 9, this IR sits between a frontend and a backend. It allows compilers developers to only focus on a specific part of the compiler. For example, writing LLVM IR optimizations does not require interacting with the frontends. These optimizations will then work regardless of the frontend.

LLVM's tools being partitioned and having well-defined APIs allows developers to easily reuse them in different projects — projects not necessarily restricted to compilers. In our case, we will be able to take advantage of LLVM's optimizations without having to interact with the frontend or backend.

5.1 The design of LLVM IR

LLVM IR is specified in the LangRef and is not necessarily backwards compatible. We decided to go with LLVM version 17.[4]

5.1.1 Abstraction

LLVM IR is a relatively high level language when compared to AMD64. Here are a couple of included niceties:

- LLVM IR has an infinite amount of registers denoted as `%<name>`
- Rather than a size, each register has a type:
  - Integer types have arbitrary sizes covering bits `i1`, regular integers `i32` and more esoteric sizes such as `i17`. The signedness of integers is not specified in the type but rather in instructions where it is required[5].
  - Pointer types are opaque and simply designated as `ptr`
  - Many other types exist including floats, arrays, vectors and structs.
- There is no need for calling conventions as `call` instructions accept arguments for functions.

[4]This was required by our other dependencies (REMILL and KLEE).

[5]For example, the division instruction has an unsigned (`udiv`) and signed (`sdiv`) variant. An addition on the other hand is the same for signed and unsigned ints, so there is only `add`.

5.1.2 Example IR

Lets breakdown a simple LLVM function which multiplies its input by two. The code for this function is available in figure 10.

```
define i32 @times2(i32 %a) {
    %a2 = mul i32 %a, 2
    ret i32 %a2
}
```

Figure 10: An LLVM function multiplying its input by two

In this sample, we define a function `times2` that takes an `i32`[6] called `a` as an argument and returns an `i32`. In the function, we use the `mul` instruction to multiply the argument by two. We specify that we are expecting an `i32`, and we pass as operands the register `a` and the constant `2`. The result of the multiplication is stored in the register `a2`. Finally, this value is returned with the `ret` instruction.

5.1.3 Static Single Assignement (SSA)

To allow for efficient optimizations, LLVM's IR uses Static Single Assignement (SSA)[7]. In an SSA program, every variable is a constant. Instead of writing `x = x + 1` and mutating `x`, we would need to create a new constant `x2` and write `x2 = x + 1`.[8]

5.1.4 Phi nodes

If we try to convert a more complex program into SSA we might struggle to represent loops and other CFG constructs. For example, let's try to convert to LLVM IR a simple C loop such as the one in figure 11.

```
int count_to_5() {
  int i = 0;
  while (i < 5) {
    i += 1;
  }
  return i;
}
```

Figure 11: A C program counting to 5

[6] LLVM's `i32` corresponds to C's `int`
[7] To be exact, only the registers are SSA. We will see in subsection 5.1.5 that in LLVM memory is mutable
[8] In LLVM syntax: `%x2 = add i32 %x, 1`

If we naively translate this to LLVM IR we would get the incorrect program in figure 12.

```
define i32 @count_to_5() {
entry:
    %i = add i32 0, 0
    br label %while

while:
    %i2 = add i32 %i, 1
    %cond = icmp slt i32 %i2, 5
    br i1 %cond, label %while, label %return

return:
    ret i32 %i2
}
```

Figure 12: An incorrect LLVM program attempting to count to 5

Although this code is valid SSA, it is not semantically correct. The loop is never exited: each iteration it adds one to zero then compares the result to five. We are always incrementing the initial value, but we would want to increment the previously incremented value.

The solution is to use *phi nodes*. A phi node assigns a value to a register based on control flow information. Figure 13 shows a correct LLVM version of our counting program.

```
define i32 @count_to_5() {
entry:
    br label %while

while:
    %i = phi i32 [0, %entry], [%i2, %while]
    %i2 = add i32 %i, 1
    %cond = icmp slt i32 %i2, 5
    br i1 %cond, label %while, label %return

return:
    ret i32 %i2
}
```

Figure 13: A correct LLVM program counting to 5 using phi nodes.

In this new version, `i` takes the value `0` if control flow was passed from `entry` to `while`, and it takes the value of `i2` if control flow was passed from `while` to `while`. In other words, the value of the register depends on the previous block.

Phi nodes are a sneaky way of getting around SSA limitations.

5.1.5 Memory in LLVM

Another way of getting around SSA's limitations is by making use of memory. In LLVM, we can do that by making an allocation on the stack [9] with the `alloca` instruction and then using `load` and `store` instructions

[9] This is not a mistake, contrary to many other languages, LLVM only performs allocations on the stack. To perform heap allocations one needs to use libraries or compiler intrinsics

to read and modify the pointed data.

Figure 14 shows an LLVM version of our counting program that uses memory.

```
define i32 @count_to_5() {
entry:
    %i_ptr = alloca i32
    store i32 0, ptr %i_ptr
    br label %while

while:
    %i = load i32, ptr %i_ptr
    %i2 = add i32 %i, 1
    store i32 %i2, ptr %i_ptr
    %cond = icmp slt i32 %i2, 5
    br i1 %cond, label %while, label %return

return:
    %i_ret = load i32, ptr %i_ptr
    ret i32 %i_ret
}
```

Figure 14: A correct LLVM program counting to 5 using memory.

In this program, we start by allocating our pointer with the `alloca i32` instruction. We can then initialize the value with a store instruction `store i32 0, ptr %i_ptr`. This instruction sets the data, pointed by `i_ptr`, to the integer `0`.

In our loop, we don't have to worry about LLVM's SSA requirements as we are not mutating our pointer, simply the pointed value.[10]

5.2 The often misunderstood GEP instruction[11].

Pointer arithmetic is achieved in LLVM through the `getelementpointer` instruction called Get Element Pointer (GEP). Suppose we have a global array of ten bytes, and want to access the third one (which is at offset 2). We can do so using the GEP instruction. Figure 15 shows an example of such a program.

```
@arr = global [10 x i3]

define i8 @third() {
    %third_ptr = getelementptr i8, ptr @arr, i64 2
    %third = load i8, ptr %third_ptr
    ret i8 %third
}
```

Figure 15: An LLVM program retrieving the third element of an array

[10] This can be thought of like the difference between these C types: `const int *` and `int const *`. An LLVM pointer is similar to C's `int const *`.

[11] This is a play on LLVM's similarly named FAQ page, which provided only further confusion

Let's have a look at the GEP instruction from figure 15. In this instruction:

- We specify the type of the array: here we are dealing with an array of `i8`
- We specify the initial address of the array: here, the global `arr`
- Finally, we specify the offset, in our case we want the third element, so we have an offset of `2`

Figure 16 contains a schematized view of this GEP instruction.

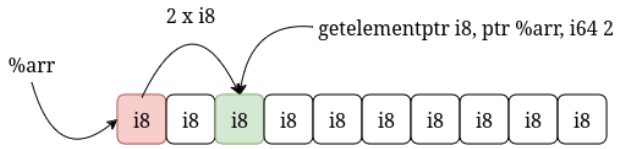


Figure 16: Schematized representation of `getelementptr i8, ptr @arr, i64 2`

Let's now suppose we are dealing with a matrix (a table of tables). We can access the element at the 2nd row (offset 1), and 3rd (offset 2) column with the code in figure 17

```
@arr = global [10 x [10 x i8]]

define i8 @get_2_3(ptr @arr) {
    %el_ptr = getelementptr [10 x i8], ptr @arr, i64 1, i64 2
    %el = load i8, ptr %el_ptr
    ret i8 %el
}
```

Figure 17: An LLVM program retrieving the element at the 2nd row and 3rd column of a matrix

The element on the 3rd row 2nd column is not necessarily equal to that on the 2nd row 3rd column, so `getelementptr [10 x i8], ptr %arr, i64 2, i64 1` would not retrieve the same address.
Figure 18 contains a schematized view of this GEP instruction.

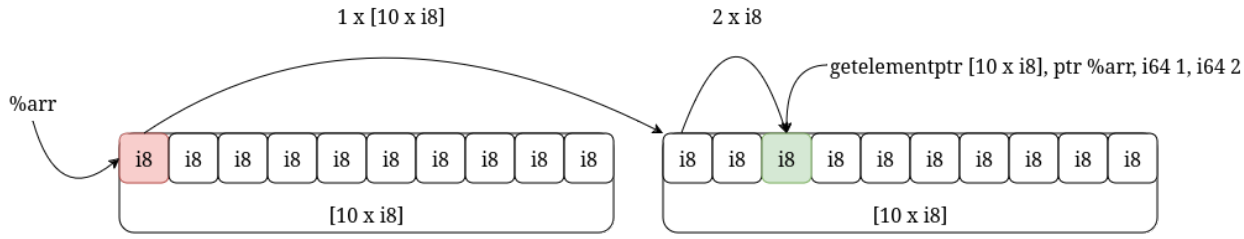


Figure 18: Schematized representation of `getelementptr [10 x i8], ptr @arr, i64 1, i64 2`

Behind the scenes, we are calculating[12]:

$$\begin{aligned} 1 \times \texttt{sizeof([10 x i8])} + 2 \times \texttt{sizeof([10 x i8][1])} &= 1 \times \texttt{sizeof([10 x i8])} + 2 \times \texttt{sizeof([1xi8])} \\ &= 1 \times 10 \times \texttt{sizeof(i8)} + 2 \times \texttt{sizeof(i8)} \\ &= 12 \times \texttt{sizeof(i8)} \end{aligned}$$

[12]These calculations are only meant to explain what the GEP instruction is doing. The `sizeof` function is not part of LLVM.

Notably, this exactly corresponds to the code produced by LLVM when using `-O3` optimizations. The original `getelementptr` instruction becomes `getelementptr i8, ptr %arr, i64 12`.

We know that the type in a GEP instruction represents the type of an element in an array. However, we can also specify the complete type of our pointer in our GEP instruction. This requires adding an offset at the beginning of the GEP instruction with a value equals to zero. This new program is available in figure 19.

```
@arr = global [10 x [10 x i8]]

define i8 @get_2_3() {
    %el_ptr = getelementptr [10 x [10 x i8]], ptr @arr, i64 0, i64 1, i64 2
    %el = load i8, ptr %el_ptr
    ret i8 %el
}
```

Figure 19: An LLVM program retrieving the element at the 2nd row and 3rd column of a matrix

Behind the scenes, we are calculating:

$$0 \times \texttt{sizeof([10x[10 x i8]])} + 1 \times \texttt{sizeof([10x[10 x i8]][0])} + 2 \times \texttt{sizeof([10x[10 x i8]][0][1])} = 0 \times 10 \times 10 \times \texttt{sizeof(i8)} + 1 \times 10 \times \texttt{sizeof(i8)} + 2 \times \texttt{sizeof(i8)} = 12 \times \texttt{sizeof(i8)}.$$

This is once again the same as $12 \times \texttt{sizeof(u8)}$.

Without the additional `0` offset, we would be calculating the following:

$$1 \times \texttt{sizeof([10 x [10 x i8]])} + 2 \times \texttt{sizeof([10 x [10 x i8]][1] = [10 x i8])}$$

...which is the same as $120 \times \texttt{sizeof(u8)}$ and not the correct offset at all!

5.2.1 Structures

LLVM also allows us to create structures using the keyword `type`. For example, `%struct.Pair = type { i32, i32 }`. In the following examples, we will use packed structs to simplify calculations. A packed struct is a struct in which no gaps are added for alignment (fields are often aligned to 4 bytes by default). In figure 20, the struct `Inner` would have a size of 48 instead of 45 (which is the length of its fields) if it wasn't packed. In LLVM, packed structs are denoted as such `%struct.Pair = type <{ i32, i32 }>`.

In C, we can easily access fields of a structure by name as shown in figure 20. In LLVM, structure fields do not have names, therefore we will have to use the GEP instruction to retrieve them.

```
struct __attribute__((packed)) Struct {
  int arr1[7];
  struct __attribute__((packed)) Inner {
    int n;
    char c;
    int arr2[10];
  } inner;
} obj;

int third_inner() { return obj.inner.arr2[4]; }
```

Figure 20: A C implementation of a function retrieving the third element of `Inner`'s array

Using the GEP pointer, we can access fields in our structure, in this case, the "offset" corresponds to the field number. Let's break down the example in figure 21.

```
%struct.Struct = type <{ [7 x i32], %struct.Inner }>
%struct.Inner = type <{ i32, i8, [10 x i32] }>

@obj = external global %struct.Struct, align 1

define i32 @third_inner() {
    %third_ptr = getelementptr %struct.Struct, ptr @obj, i32 0, i32 1, i32 2, i32 4
    %third = load i32, ptr %third_ptr
    ret i32 %third
}
```

Figure 21: An LLVM implementation of a function retrieving the third element of `Inner`'s array

When using structures, the GEP instruction is used to retrieve a particular field in a structure. For example, the `Inner` structure is the second field in `Struct` so it is accessible at offset `1` (we still index starting at 0). Similarly, the field `arr2` is the third field of `Inner` so it can be accessed at offset `2`.

Behind the scenes, we are calculating

$$\begin{aligned}(0 \times \texttt{sizeof(Struct)}) + (\texttt{sizeof(Struct[0])}) + (\texttt{sizeof(Struct[1][0])} + \texttt{sizeof(Struct[1][1])}) \\ +(4 \times \texttt{sizeof(Struct[1][2][0])}) = \\ (0 \times \texttt{sizeof(Struct)}) + (\texttt{sizeof([7 x i32])}) + (\texttt{sizeof(i32)} + \texttt{sizeof(i8)}) + (4 \times \texttt{sizeof(i32)}) = \\ (0 \times 73 \times \texttt{sizeof(i8)}) + (28 \times \texttt{sizeof(i8)}) + (5 \times \texttt{sizeof(i8)}) + (16 \times \texttt{sizeof(i8)}) = \\ 49 \times \texttt{sizeof(i8)}\end{aligned}$$

Once again, this is the byte offset we find with `-O3` optimization level: `getelementptr i8, ptr @obj, i32 49`.

6 Initial analysis

In this first part, we try to understand what obfuscation through virtualization looks like. Compared to the state of the art, this is a technical hands-on illustration.

6.1 Manual deobfuscation

Before diving into automated deobfuscation tools, it is important to understand the alternative: manual deobfuscation. This experiment will allow us to understand if automated tools are necessary in the first place. It will also allow us to familiarize ourselves with the TIGRESS obfuscator, and the kind of obfuscated code it produces.

As a case study, we decided to manually deobfuscate `challenge0-1` from the TIGRESS challenges [51], which is a good example of a binary that features a simple virtual machine with no additional obfuscation. These challenge programs were generated randomly, so we should not expect to recognize a well-known algorithm after deobfuscation.

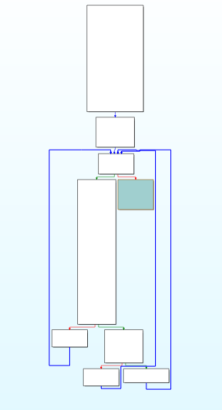


Figure 22: No handlers in IDA

Upon loading the binary into IDA, we notice that it struggles to resolve the indirect jump in the dispatcher. As discussed in section 1.1.3, indirect jumps are a challenge for static analysis. Figure 22 shows a zoomed out view of the obfuscated function in IDA. The blue block represents the VM dispatcher. As explained in section 2.7, the dispatcher is the parent block of each handler and therefore should have many children (a VM often has multiple handlers). However, in our example, IDA was unable to find any.

By manually adding the handlers to the function, we are able to create a somewhat more accurate CFG (Figure 23). However, the dispatcher (colored red in Figure 23) is still not connected to the handlers. Although we found the jump table and all the handlers, they are not standard enough for IDA to link them together.

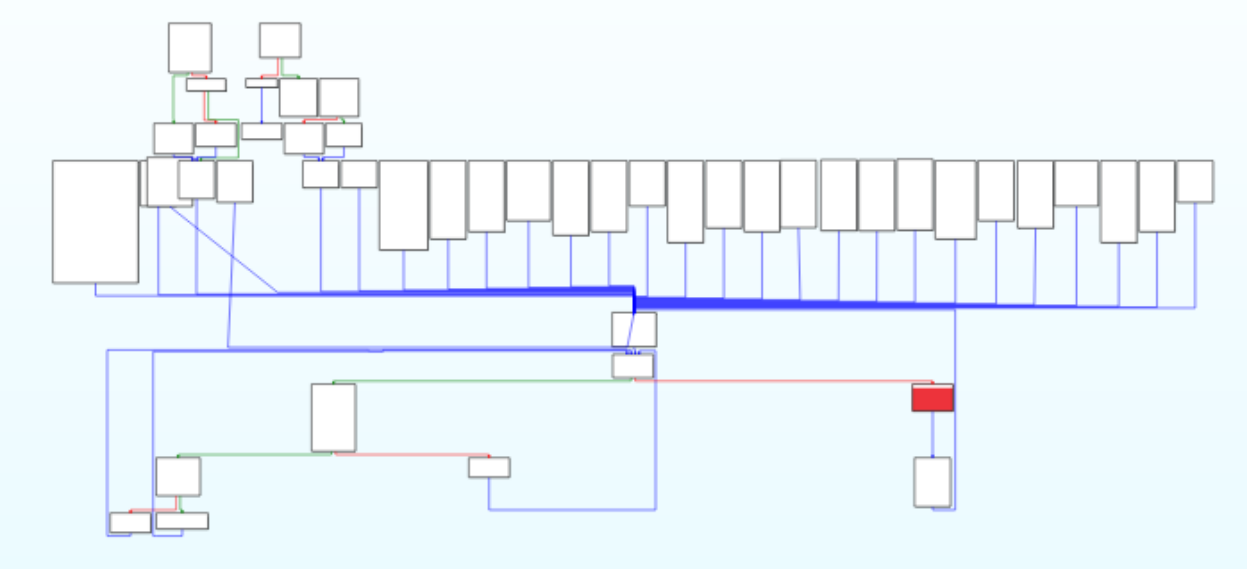


Figure 23: `challenge0-1` CFG in IDA, in red the dispatcher

Despite the missing edges, with this CFG, we can now attempt to deobfuscate this function. The strategy is as follows: locate the jump table and virtual instructions, deobfuscate each handler, then create a script

to decompile the virtualized program.

If we take a look at a handler (see Listing 24a), we notice that the VM operates with a virtual stack, pointed by `s_p`. Most handlers operate by retrieving their arguments from the stack, performing an operation, then saving the result back to the stack.

The control flow is handled with a Virtual Program Counter (VPC) (`i_p`) incremented in most handlers (see Listing 24a). This program counter is used as an offset in the array of obfuscated instructions to retrieve an instruction code. A binary search then matches each instruction code to a handler by searching in the "jump table" (Figure 24b).

```
add_8:
    ; Increment the virtual
    ; instruction pointer i_p
    mov     rax, [rbp+i_p]
    add     rax, 1
    mov     [rbp+i_p], rax

    ; Store the address where to push
    ; the result on virtual stack
    mov     rax, [rbp+s_p]
    lea     rdx, [rax-8]

    ; Fetch the right operand
    ; (top of virtual stack)
    mov     rax, [rbp+s_p]
    mov     rcx, [rax]

    ; Fetch the left operand
    ; (second word of virtual stack)
    mov     rax, [rbp+s_p]
    sub     rax, 8
    mov     rax, [rax]

    ; The actual addition
    add     rax, rcx

    ; Push the result
    ; (second word of virtual stack)
    mov     [rdx], rax

    ; Pop the virtual stack
    mov     rax, [rbp+s_p]
    sub     rax, 8
    mov     [rbp+s_p], rax

    jmp     top_of_loop
```

(a) The assembly code for a handler performing an addition

```
; KeyValueStruct jump_table[28]
jump_table      KeyValueStruct <3, offset jump_29>
                KeyValueStruct <9, offset push_local_7>
                KeyValueStruct <0Ch, offset add_8>
                KeyValueStruct <17h, offset as_signed_15>
                KeyValueStruct <1Bh, offset add_9>
                KeyValueStruct <1Eh, offset cmp_21>
                KeyValueStruct <30h, offset add_10>
                KeyValueStruct <37h, offset dderef_28>
                KeyValueStruct <3Fh, offset _sub_20>
                KeyValueStruct <43h, offset ddref_25>
                KeyValueStruct <45h, offset load_int_5>
                KeyValueStruct <4Fh, offset deref_short_4>
                KeyValueStruct <50h, offset IO_27>
                KeyValueStruct <5Fh, offset ref_assign_6>
                KeyValueStruct <6Bh, offset NOOP_22>
                KeyValueStruct <77h, offset and_2>
                KeyValueStruct <82h, offset xor_19>
                KeyValueStruct <90h, offset jnz_23>
                KeyValueStruct <95h, offset load_int_11>
                KeyValueStruct <9Fh, offset ret_24>
                KeyValueStruct <0B8h, offset DEREF_26>
                KeyValueStruct <0BDh, offset deref_1>
                KeyValueStruct <0C2h, offset load_int_13>
                KeyValueStruct <0C5h, offset lshift_14>
                KeyValueStruct <0C8h, offset add_quad_17>
                KeyValueStruct <0C9h, offset shr_3>
                KeyValueStruct <0CFh, offset mul_16>
                KeyValueStruct <0DFh, offset str_word_18>
                KeyValueStruct <0F2h, offset or_12>
_data           ends
```

(b) The jump table for the obfuscated program in IDA

With all this information, we can now build a Python script to disassemble our virtualized program. We chose to write handlers outputting pseudo-C. Figure 25 shows a sample Python handler for the custom

```python
# add
def handler_8():
    e1 = stack.pop()
    e2 = stack.pop()

    stack.append(f"({e1} + {e2})")
    return execute_next()
```

Figure 25: A python handler decoding an addition

disassembler. Figure 26 shows a sample of the deobfuscated program, after some manual touch-ups (variable naming, instruction grouping).

```
v16[0] = (input[0] & 0x222c2afc) - 0x14582014;
v16[1] = (0x1df2339f * v16[0]) + 0x608c69 + 0x22722b13 + input[0];
v16[2] = (input[0] & 0x140538e4) - 0x5f1e4ce7;
v16[3] = (v16[1] >> (((v16[2] >> 0x3) & 0x7) | 0x1)) + input[0];
if ((v16[0] ^ v16[1]) == (v16[3] * v16[2])) {
  v16[0] = (v16[0] | ((v16[1] & 0x7) << (0xff & 0x2)));
} else {
  ...
```

Figure 26: A sample of the deobfuscated program

The final code is readable[13]. The deobfuscated C code was compiled, and we tested that the input/output behavior of the obfuscated and deobfuscated binaries matched on a couple values.

Overall, this experiment was quite long and tedious. It took me about two days, and it took Sebastian Millius (the original solver) roughly 8 hours [51] for all 5 challenges[14]. Although this might seem like a reasonable amount of time, it is important to keep in mind that these results were achieved for an extremely simple program (with a single if statement), an equally simple instruction set, and with only virtualization as obfuscation. Obfuscated programs often use multiple layers of obfuscation, on much larger programs.

Nevertheless, this experiment provided us with some valuable information:

- Rebuilding CFGs is hard.
- Manual deobfuscation of virtualized binaries is tedious.
- We now have a deobfuscated version of `challenge0-1` (this is not provided by Tigress).

This analysis also allowed us to understand what parameters were used during obfuscation. Indeed, the command provided with the challenge is incomplete, with some parameters such as `VirtualizeDispatch` hidden.

```
tigress --Verbosity=1 --Seed=$seed --FilePrefix=obf --Transform=Virtualize \
    --Functions=SECRET --VirtualizeDispatch=binary --VirtualizeOperands=stack \
    --out=$output.c $input.c
```

Figure 27: Command used to obfuscate the `challenge0-1`

[13] Although it does look like gibberish because it was randomly generated.
[14] It took him 4 hours to create a deobfuscation script, and then 4 hours to adapt it to the 5 challenges and debug it.

We can now say that the parameter `VirtualizeDispatch` was passed the value `binary`, indicating that a binary search was used to match instruction codes to handler addresses. The complete obfuscation command is available in figure 27.

6.2 Automatic deobfuscation strategy

The length and difficulty of the manual deobfuscation experiment justifies the need for automated deobfuscation tools. Out of the different automated deobfuscation strategies highlighted in part I.3, we decided to follow Triton's approach. Triton [45, 42] was chosen because it was used to solve 4 out of the 6 Tigress challenges, and we wanted to see if we could reproduce these results.

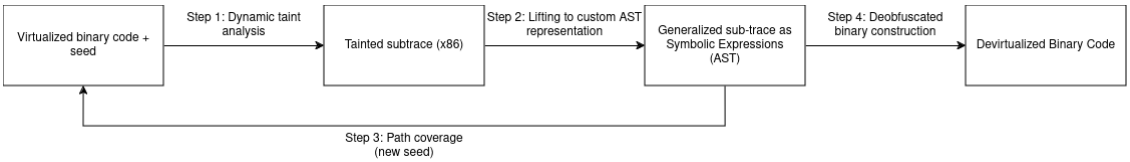


Figure 28: Triton schematized approach

As a reminder, the strategy used by Triton (summarized in figure 28) is as follows. First, it collects tainted sub-traces of the obfuscated program with different seeds. These seeds are computed using symbolic execution to maximize coverage. Then, it reconstructs and optimizes this set of sub-traces to deobfuscate the binary. We will explain how this is achieved in further detail during this report.

6.3 Limits of Triton for automatic deobfuscation

To better understand Triton's approach, we downloaded the tool and used it to partially deobfuscate some Tigress challenges.

Using the Triton library, we were able to partially deobfuscate `challenge0-1`. We did not use the provided script but rather attempted to rewrite our own using the Triton library and our understanding of the paper. Triton gave us very good results, but they were quite slow: taint analysis alone took around 20s on my machine. In [43], Jonathan Salwan *et al.* claim to solve the challenge in 9s, which is very much in line with my results — we have the same order of magnitude, and I expected their script to be faster than mine.

The key takeaway for speed is that Triton can take up to ten seconds to deobfuscate very simple programs (*challenge0-1* is a simple if/else statement). We believe one reason for this is that Triton relies heavily on symbolic execution which is notoriously slow.

If we want to deobfuscate much larger and more complex programs, we need a faster tool.

Another limitation of Triton we hope overcome, is that Triton uses an internal AST representation making it difficult to interoperate with other existing tools.

6.3.1 A hypothesis: starting with optimizations

Alongside these limitations, we had a slight improvement idea: why not put optimizations at the beginning of our deobfuscation rather than at the end?

Adding this step could reduce analysis time: less code means less code to analyze means faster run times.

Although this was an interesting idea which motivated the creation of a new tool, we did not have time to test this hypothesis during the internship. The optimization is present, but for it to make a difference, we would need more obfuscated binaries which we did not have time to test on.

7 Mermaid: an automatic deobfuscation tool using LLVM

The limitations we discussed with Triton motivated the creation of a new tool: Mermaid (sticking with the sea people theme). This new tool is designed to be a foundation to deobfuscate much larger binaries. As such, special care was given to performance and modularity. Rather than creating one monolithic tool, we built a suite of small interoperating tools, each relying heavily on LLVM.

LLVM was chosen as it offers powerful optimizations and is used in many reverse engineering projects as an IR. We hope that by using LLVM as an IR, our tool will better interoperate with existing tools.

During this internship, a large part of Triton's deobfuscation strategy was implemented in this new tool. However, we did not have time to implement a loop that calculates new seeds to improve coverage as done by Triton.

7.1 Deobfuscation strategy

Although we have mostly mentioned Triton, our deobfuscation strategy is in reality much closer to the work done by Yadegari *et al.* [57]. Our deobfuscation tool is going to be a lot simpler than Triton or the tool written by Yadegari *et al.* [57], however we will be taking advantage of LLVM. Although Triton also makes use of LLVM, we plan to place LLVM at the center of our tool.

Figure 29 shows the overview of how Mermaid deobfuscates binaries.

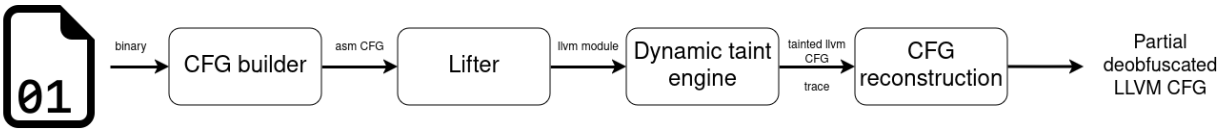


Figure 29: Mermaid's deobfuscation approach

7.2 The obfuscated program

During the development of Mermaid, we used the Collatz flight time as our test program (figure 30).

The Collatz conjecture states that the sequence (u_n), described in equation 6, will eventually reach 1 for any value $u_0 \in \mathbb{N}$.

$$\forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \equiv 0 \mod 2 \\ 3 \times u_n + 1 & \text{if } u_n \equiv 1 \mod 2 \end{cases} \tag{6}$$

We chose Collatz as it has a simple yet interesting CFG. It features both a loop and a conditional branch. Furthermore, we can easily fit Collatz into our Tigress challenge template.

The advantage of using Collatz rather than `challenge0-1` is that we have complete control over the code, the compilation and obfuscation steps. This amount of control, not available in `challenge0-1`, simplifies analysis and debugging making it more suited for developing tools.

```c
void SECRET(uint64_t *input, uint64_t *outputs) {
  uint64_t x = input[0];
  outputs[0] = 0;

  while (x != 1) {
    if (x % 2 == 0) {
      x = x / 2;
    } else {
      x = 3 * x + 1;
    }
    outputs[0] += 1;
  }
  outputs[0] *= 3;
}
```

Figure 30: Our Collatz inspired `SECRET` function

This program was then obfuscated using the Tigress obfuscator version `3.1`, using the same command as that used in the 0th-Tigress challenges (figure 27).

7.3 Building a CFG

The first step in deobfuscating a binary consists in retrieving the CFG of the obfuscated function from the binary.

7.3.1 Challenges with static analysis

As previously stated in the state of the art (cf I.1), extracting CFGs is not a simple task. We expect our obfuscated function to make use of an indirect jump in the dispatcher. The presence of an indirect jump greatly complicates any complete static analysis. IDA not resolving the CFG is also a good indicator that we should not use only static analysis. Instead, we will opt for a dynamic reconstruction of a partial CFG.

7.3.2 The motivation behind partial CFGs

During dynamic reconstruction of a CFG, we are not guaranteed to retrieve a complete CFG. There might exist multiple execution paths, and we only visit one of them. Take the simple example of a program with two execution paths. We pick the second execution branch if the input (in figure 31, `qword ptr [rdi]`) matches a constant (in figure 31, `0xdeadbeef`). It is unlikely that we will guess this constant, and our execution will likely follow the first execution path.

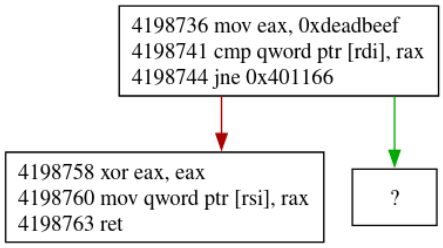


Figure 31: Partial x86 CFG showing a missing block

Figure 31 shows an example partial CFG. Although we did not explore each execution path, we did visit the CFG instruction leading to another path. We are therefore able to indicate that at least one execution path is missing. This is represented as a block with a question mark (the missing execution path could be composed of multiple blocks — the box is purely visual).

7.3.3 Dynamic partial CFG construction

To rebuild the partial CFG dynamically, we started by emulating the program using Unicorn [14] in order to retrieve an execution trace. We chose emulation as it seemed like a simpler strategy. However, other more efficient strategies could be used to retrieve a trace including *Intel Processor Trace* [23].

Once we had obtained a trace, we designed a simple algorithm to reconstruct a CFG from a trace (figure 1).

Algorithm 1 CFG from trace

Require: An execution trace *trace*
Require: A partial CFG *cfg*
  block ← create empty block with no parent
  for each *instruction* in the *trace* **do**
    block[*block.size*] ← *instruction*
    block.size ← *block.size* + 1
    if *instruction* is a control flow instruction **then**
      parent ← INSERT(*cfg*, *block*)
      block ← create block with parent *parent*
    end if
  end for

The more complicated part of the algorithm is the insertion of blocks into our CFG (algorithm 2). Indeed, as we split blocks on control flow instructions, our blocks could overlap with pre-existing blocks in our CFG. The insertion algorithm guarantees that instructions appear exactly once in the CFG.

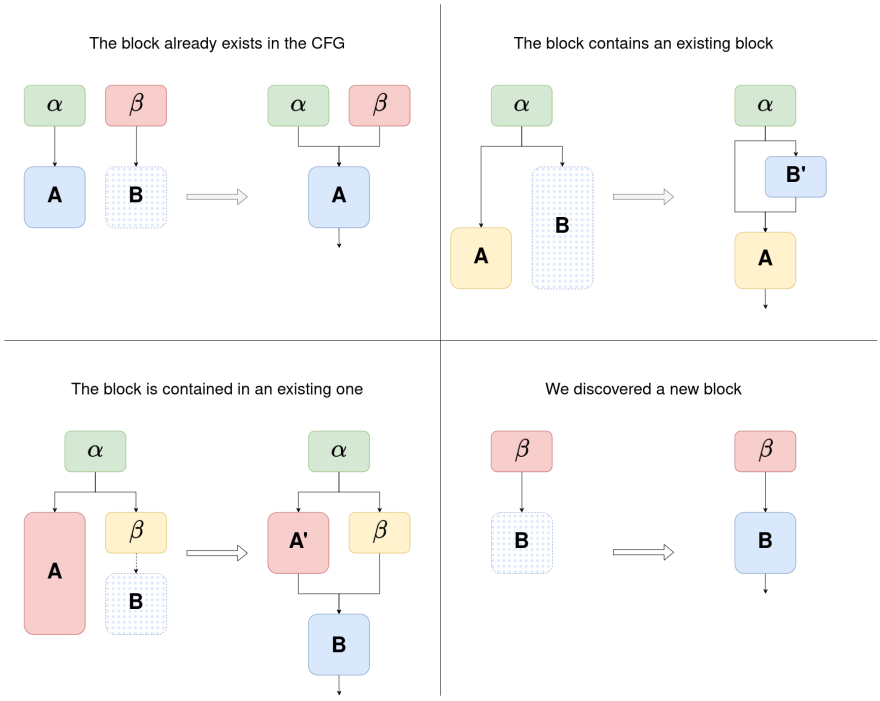


Figure 32: Different conditions encountered by the block insertion algorithm

7.3.4 Results

Using this algorithm, we are able to rebuild a partial CFG. This CFG is only partial as our trace used a single input value, other execution paths could exist for different values. For example, in figure 31 our CFG is missing a path that is explored only if the input equals `0xdeadbeef`.

Multiple strategies exist to improve the completeness of our CFG. We could use symbolic execution to retrieve input values to explore other paths (the strategy used in TRITON). We could also perform a static

Algorithm 2 Inserts a block B into a partial CFG *cfg*

Require: Block B
Require: A partial CFG *cfg*
if $\exists A \in cfg$ such that $A[A.size] = B[B.size]$ **then**
    if $B.size = A.size$ **then** ▷ The blocks are identical
        $A.parents \leftarrow A.parents \bigcup B.parents$
        return A
    else if $B.size < A.size$ **then** ▷ A contains B
        Remove the last $B.size$ addresses from A
        Add B to *cfg*
        for $C \in cfg$ such that $A \in C.parents$ **do**
            $C.parents \leftarrow C.parents \setminus A$
            $C.parents \leftarrow C.parents \bigcup \{B\}$
        end for
        $B.parents \leftarrow B.parents \bigcup \{A\}$
        return B
    else ▷ B contains A
        Remove the last $A.size$ addresses from B
        $B \leftarrow$ INSERT(*cfg*, B)
        $A.parents \leftarrow A.parents \bigcup \{B\}$
        return A
    end if
else ▷ B is a new block
    Add B to *cfg*
    return B
end if

analysis to explore blocks at discovered but unexplored addresses (like in the example in figure 31: we know that there is an unexplored block starting at `4198744 + 2 = 0x40115a`).

However, for this internship, these partial CFGs are sufficient. This is because from our testing, all the blocks in the obfuscated binary get visited with any input.

We provided in the appendix both the partial x86 CFG of our Collatz function (appendix A.1) and that of our obfuscated Collatz function (appendix A.2).

It took 15ms to emulate and build a CFG. The original Collatz program on the same input took 0.5ms.[15]

7.3.5 LibCFG

Throughout this work, we are working with a lot of CFGs. We often want to be able to perform common operations (loading, storing, displaying) on CFGs from multiple languages.

After considering other binary exporters (Quokka [37], BinExport [4], ...), we could not find an adapted tool. Indeed, for this project, we need very accurate CFG and instruction information that is language independent. Meanwhile, we do not need any information on data sections, symbols etc. Moreover, we need a representation that can support both AMD64 and LLVM IR.

For all these reasons, we decided to build LibCFG. LibCFG is a C++ library for interacting with CFGs. The CFGs can be saved to and loaded from a binary using FlatBuffers, or displayed using Graphviz.

In order to build this library and ensure multi-language support, we had to distill what makes up a CFG. To remain as general as possible, we defined a CFG as a directed graph, where each node (called Block) is composed of a sequence of Instructions. Blocks can have multiple parents and multiple children. Instructions

[15] These results need to be taken with a grain of salt, as they represent a couple executions of a single program. In general, execution times in this report are given as an order of magnitude. In this situation, the key takeaway is that CFG reconstruction is very quick for an analyst (under 1s).

contain addresses, but other than that have no special requirements. We make no assumption on addresses (no uniqueness, no need to be raising nor continuous). Finally, a CFG must have exactly one entrypoint (a block with no parents).

Using this definition, we built LibCFG adapters for AMD64, LLVM and a generic string language.

We used LibCFG to generate most of the CFGs in this report. However, our program still struggles to display edge colors properly, therefore you might notice miscolored edges in the graphs. This is purely an aesthetic issue.

7.3.6 Typing a CFG

During this project, we stumbled upon CFGgrind [41]. CFGgrind, is a tool that dynamically builds CFG from traces in Valgrind [33].

An interesting concept put forward in CFGgrind is the notion of instruction types. They proposed five types of instructions:

- *standard*: flows to the next instruction
- *jump*: unconditional jump
- *branch*: conditional branch
- *call*: calls a function
- *return*: transfers control back to the caller

We made multiple changes to their design to incorporate it to LibCFG. First, we removed the *call* type as it does not fit in our scope.

Second, instead of adding types at the instruction level, we added them to blocks. In a block, every instruction prior to the last one is necessarily a *standard* instruction (or a *call*, but we removed those). We therefore use the CFGgrind type of the last instruction as the LibCFG type of the block.

Our type system has six block types:

- *Standard*: flows to the next instruction
- *Jump*: unconditional jump to a known address
- *IndirectJump*: unconditional jump to an unknown address
- *Branch*: conditional branch to a known address
- *IndirectBranch*: conditional branch to an unknown address
- *Return*: transfers control back to the caller

These types are generic enough to have a meaning outside any specific language. For example, the *Jump* type can be used to represent a basic block ending with a `BR` instruction in LLVM, a `JMP` instruction in AMD64 or a `B` instruction in ARM.

7.4 Lifting to LLVM

The next step in our deobfuscation journey consists in lifting our program to LLVM IR. In the previous part, we managed to extract an AMD64 CFG from a binary. In this step, we want to convert this AMD64 CFG to an LLVM IR one.

7.4.1 Advantages of code lifting

Lifting a binary to an intermediate representation can simplify the analysis step for multiple reasons.

First, the design of the IR can be more adapted for analysis. In our case, LLVM's IR uses SSA which greatly simplifies optimizations. It also simplifies more complex instructions (such as AMD64's `AESDEC`).

IRs also provide a platform-independent representation of the program. After translation, we can therefore leverage all our analysis passes regardless of the original architecture (AMD64, ARM, ...).

Finally, IRs, and in particular LLVM's IR, provide a standard that multiple tools can use to communicate between each other.

7.4.2 Over-expressiveness

Lifting is another difficult step. This might be surprising as during compilation, LLVM IR is translated to AMD64, so why is the reverse operation difficult?

Most AMD64 instructions encode multiple behaviors. For example, a `SUB` instruction performs both a subtraction and a comparison (by setting flags). Although a compiler will usually prefer the `CMP` instruction for comparisons, it is also possible for the `SUB` instruction to be used instead. However, most times, the `SUB` instruction is used just as a subtraction.

The compiler gets to pick which and how many behaviors of an instruction are used[16]. However, given the binary, it is hard to know which behavior is being used. This explains why our lifter always needs to translate each behavior of each instruction.

However, describing the behavior of each instruction in excruciating details may lead to over-expressiveness. Over-expressiveness occurs when the size of a program significantly expands after being lifted to an intermediate representation.

7.4.3 REMILL

We chose to use REMILL [39] as a base for our lifting. REMILL is a popular tool for lifting various instructions to LLVM. It was built by Trail of Bits for their tool McSema [31].

REMILL works by describing the semantics of each instruction in C++. For example, figure 33 shows the C++ function describing the AMD64 `ADD` instruction. The advantage of using C++ is that the developer has access to multiple functions to handle recurring operations (for example `Read` to read an operand). The second advantage is the ability to use C++'s template meta-programming to describe multiple operations in a single function. In this example, `ADD` describes the semantics of the instruction reading from memory, registers or immediate (handled through the `S1` and `S2` types).

```
template <typename D, typename S1, typename S2>
DEF_SEM(ADD, D dst, S1 src1, S2 src2) {
  auto lhs = Read(src1);
  auto rhs = Read(src2);
  auto sum = UAdd(lhs, rhs);
  WriteZExt(dst, sum);
  WriteFlagsAddSub<tag_add>(state, lhs, rhs, sum);
  return memory;
```

Figure 33: REMILL's description of the AMD64 instruction `ADD` in C++

This C++ code is then compiled into LLVM IR using Clang. All these definitions are stored in an LLVM module located at `<install-location>/share/remill/17/semantics/x86.bc`. In this module we found multiple definitions for our `ADD` including operations with ints, memory and floats.

Finally, REMILL is able to lift code by "stitching" together these definitions.

[16] The other behaviors are simply ignored, for example the flags after a `SUB` instruction might be overwritten

To describe instruction semantics, REMILL makes use of a special structure called `%state`. This structure represents the internal state of the CPU and is architecture dependent. REMILL also uses a void pointer called `%MEMORY` to perform all operations that happen in memory. These variables can be seen in action in figure 34. This code pushes `RDI` on the stack. The pointers to the registers are retrieved from the `%state` pointer earlier.

```
%RDI = getelementptr inbounds %struct.State, ptr %state, i32 0, i32 0, i32 6, i32 11, i32 0, i32 0
%RSP = getelementptr inbounds %struct.State, ptr %state, i32 0, i32 0, i32 6, i32 13, i32 0, i32 0
...
push_rdi:
    %pc = load i64, ptr %NEXT_PC, align 8
    store i64 %pc, ptr %PC, align 8
    %next_pc = add i64 %pc, 5
    store i64 %next_pc, ptr %NEXT_PC, align 8
    %rsp = load i64, ptr %RSP, align 8
    %store_addr = sub i64 %rsp, 112
    %store_value = load i64, ptr %RDI, align 8
    %mem = load ptr, ptr %MEMORY, align 8
    %new_mem = call ptr @__remill_write_memory_64(ptr %mem, i64 %store_addr, i64 %store_value)
    store ptr %new_mem, ptr %MEMORY, align 8
    br label %next_instruction
```

Figure 34: Lifted output for the AMD64 instruction `PUSH RDI`

Despite all of REMILL's hard work, the lifting is not yet done. Indeed, REMILL is not a lifting tool per se: instead, it describes itself as "a library for other tools". Out of the box, REMILL's results may seem underwhelming: the generated IR is extremely large, many functions lack definitions and LLVM is seemingly unable to optimize the program. We ran into the issue of over-expressiveness.

To give some idea of scale, a single `ADD` instruction lifted from AMD64 to LLVM IR with REMILL yields a 132 line long module! The LLVM `add` instruction could have been sufficient (granted we are not interested in various side effects of AMD64's `ADD` instruction, for instance the carry flag).

7.4.4 REMILL intrinsics

REMILL makes extensive use of small undefined functions called intrinsics. [17] In REMILL's vernacular, intrinsics are pieces of logic left to the user of the library:

> REMILL models the semantics of instruction logic and its effects on processor and memory state, but it does not model memory access behaviors or certain types of control flow. REMILL defers the "implementation" of those to the consumers of the produced bitcode.

REMILL does not provide any default implementation or documentation for these functions. Understanding the exact meaning and use case behind each intrinsic proved to be a significant pain point during a large part of the internship.

Some examples are easy to understand, such as the `__remill_read_memory_8` provided in figure 35 (note the use of the global variable RAM). Other times, intrinsics are only used as markers, such as `__remill_jump`.

[17] These intrinsics differ from compiler intrinsics.

```
@RAM = external global [0 x i8]

; MEMORY ACCESS
define dso_local i8 @__remill_read_memory_8(i8* noundef %memory, i64 noundef %offset) {
    %1 = getelementptr [0 x i8], [0 x i8]* @RAM, i64 0, i64 %offset
    %value = load i8, i8* %1, align 1
    ret i8 %value
}

; MARKER
define dso_local ptr @__remill_jump(ptr %state, i64 %pc, ptr noundef %mem) {
    ret ptr %mem
}
```

Figure 35: Intrinsic definitions for Remill

In Saturn [21], intrinsics were defined in C++ and compiled to LLVM IR. For Mermaid, we hand-wrote each intrinsic in LLVM IR, the rationale being that the resulting shorter intrinsics would be better optimized.[18]

7.4.5 Saturn

In our research, we came across a paper on deobfuscation tool using LLVM called Saturn. Saturn is built on top of Remill and its authors came up with many creative solutions to Remill's shortcomings.

Although Saturn is not public, we were able to reach out to the authors, who gave us many insightful tips. The advice not only included general architecture choices but also specific LLVM tricks. Most notably, they helped us reimplement some of their clever LLVM passes.

One of the many tricks in Saturn is the use of an LLVM global *RAM*, rather than the default *memory* pointer. It being a global value, rather than a pointer overwritten at each instruction, allows LLVM to perform better alias analysis.[19]

A big thank you to Matteo and Peter for their help and kindness.

7.4.6 Recreating a CFG

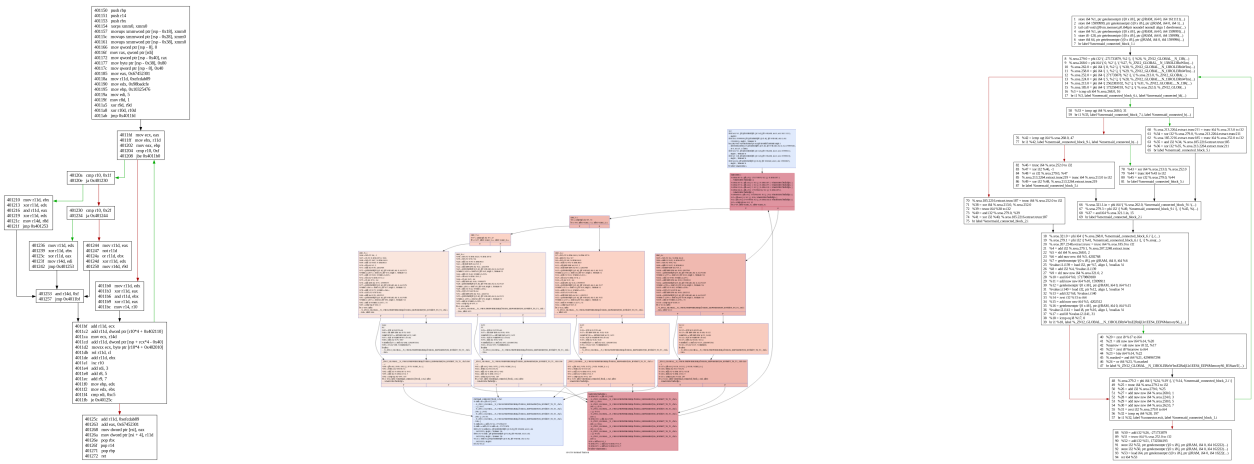
Remill is quite good at lifting individual instructions, but to lift an entire CFG we need to do some additional work.

In Saturn, each basic block is lifted to its own function. Then, a different function (the *control flow function*) is used to "call" the blocks in the correct order. This is done to allow simple optimizations on the block level. Furthermore, the authors take advantage of the simplicity of the *control flow function* to easily manipulate the CFG. We tried two strategies inspired by Saturn.

The first strategy consisted in having multiple functions call each other. The call graph of the lifted program would match the CFG of the original program. Our hope was that by inlining all the functions with LLVM we would (in theory) retrieve the original CFG. This approach was motivated by the fact that it does not require messing with LLVM's *PHI nodes*. However, with this strategy, LLVM duplicated many blocks. The new CFG did not look like the original CFG. In figure 36, we can see how the call strategy (figure 36b) duplicated blocks.

[18] We did not test it.

[19] We did not directly test this. But we did obtain much better results after using all of Saturn's advice.



(a) Original AMD64 CFG

(b) Lifted CFG with call strategy (using LLVM CFG generator)

(c) Lifted CFG with jump strategy

Figure 36: CFGs for the MD5 algorithm

The second strategy consisted in creating a skeleton function with the proper CFG. This skeleton function performs a call in each basic block to a function containing the lifted basic block. Figure 37 shows the beginning of a skeleton function. The control flow is handled using the value of the program counter (in figure 37 `%RIP_1`, which is then loaded in `%pc_after_inner_1`):

- If the block has only one child (for example, in figure 37: `mermaid_connected_block_0`), we jump to the child.
- If the block has multiple children (for example, in figure 37: `mermaid_connected_block_1`), we switch on the value of the program counter and jump to the correct block.

Our hope is that during LLVM's constant propagation pass, the program counter gets concretized and LLVM is able to simplify the CFG. Indeed, we know the value of the CFG at the beginning of a block (in figure 37, 4198726[20] at the beginning of `mermaid_connected_block_0`), all LLVM has to do is propagate this constant throughout the block. Experimentally, we were able to verify that LLVM is able to concretize the program counter. And the resulting lifted CFG is very similar to the initial one (as seen in figure 36c).[21]

[20] This address is represented as a decimal in the LLVM code.

[21] You might notice that the LLVM function has an additional block. We will discuss this later.

```
define internal ptr @skeleton(ptr %state, i64 %input, ptr %mem) {
    entry:
        br label %mermaid_connected_block_0

    default:
        unreachable

    mermaid_connected_block_0:
        %mem_0_in = phi ptr [ %mem, %entry ]
        %mem_0 = call ptr @mermaid_inner_block_0(ptr %state, i64 4198726, ptr %mem_0_in)
        %RIP_0 = getelementptr inbounds %struct.State, ptr %state, 69420
        %pc_after_inner_0 = load i64, ptr %RIP_0, align 8
        br label %mermaid_connected_block_5

    mermaid_connected_block_1:
        %mem_1_in = phi ptr [ %mem_5, %mermaid_connected_block_5 ], [ %mem_2, %mermaid_connected_block_2 ]
        %mem_1 = call ptr @mermaid_inner_block_1(ptr %state, i64 4198818, ptr %mem_1_in)
        %RIP_1 = getelementptr inbounds %struct.State, ptr %state, 69420
        %pc_after_inner_1 = load i64, ptr %RIP_1, align 8
        switch i64 %pc_after_inner2, label %default [
            i64 4198847, label %mermaid_connected_block_6
            i64 4198805, label %mermaid_connected_block_7
        ]
...
```

Figure 37: LLVM skeleton function

We decided to use an approach based on the program counter as we did not find a way to work directly inside Remill's lifted blocks.

7.4.7 Wrappers

We then added wrappers to concretize certain values before our lifted program. In Saturn, concretization is done using an optimization pass, but that seems excessive. We want to take advantage of the fact that inlining small functions in LLVM is very easy.

To do so, we created generators for small functions that concretize certain registers (for example in Amd64 we might want to concretize `RSP`, `RBP`, `FS`, `DF`).

We also created a simple wrapper for our function type. As we know that our function takes a single integer value as argument and outputs another integer, we created a wrapper that changes our function prototype to one that takes in an integer by value and returns an integer (going from `void SECRET(int* input, int*output)` to `int SECRET(int input)`). Having less memory operations means LLVM's aliasing will work better in turn allowing for better optimizations.

7.4.8 Result

Figure 38 provides an overview of the different steps involved in lifting an AMD64 CFG.

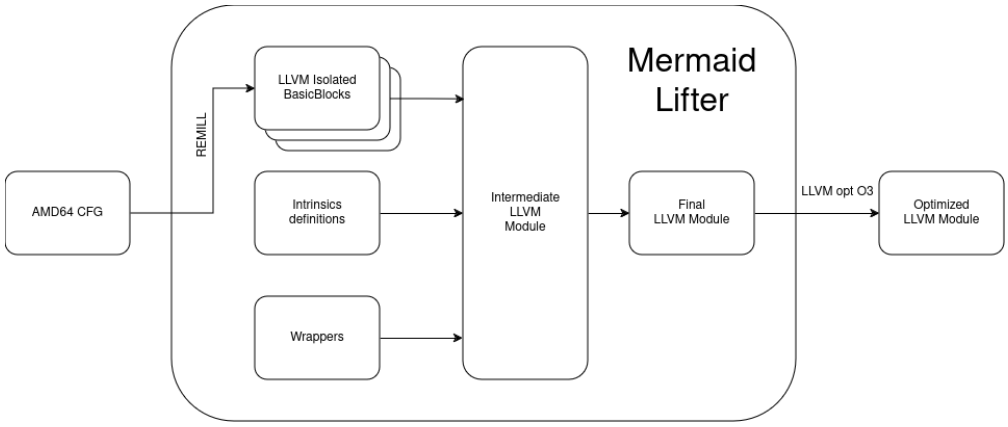


Figure 38: Overview of MERMAID lifter architecture

We also included the LLVM CFG of the obfuscated Collatz function in appendix A.2, and a smaller version in figure 39.

7.4.8.1 Execution time Lifting our obfuscated Collatz function took a little under 100ms. The resulting LLVM module has 4.5K lines.

7.4.8.2 Optimizations After running `opt` with `-O3` (which took 200ms), the resulting module is 400 lines long.

7.4.8.3 CFG sizes Our AMD64 CFG had 147 instructions while the lifted LLVM one has 220 (a 50% increase).

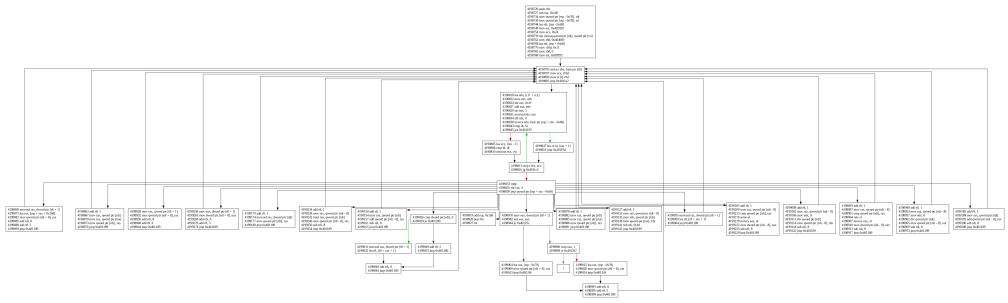


Figure 39: Partial lifted CFG of the obfuscated Collatz function

7.4.9 Graph Edit Distance

After building the lifter, we wanted to measure its performance. The simplest strategy consists in verifying that the input/output behavior of our lifted program matches that of the original binary on a small set of inputs. Such a test is relevant, but not enough to measure the performance of our lifter.

Indeed, we not only want the input/output behavior to be similar, we also want the control flows to remain similar. For example, loops should ideally not be lost during lifting.

However, we do expect the CFG to change a little. Indeed, certain AMD64 instructions are lifted to multiple blocks in LLVM (such as the `REP` instruction[22]). Because of that, we need to find a test that can measure how similar two CFGs are rather than simply trying to match the graphs.

As done by Yadegari *et al.* [57], we decided to compare the CFGs using the Graph Edit Distance (GED). The Graph Edit Distance is a measure of similarity between two graphs. It is defined as the smallest amount of operations required to transform one graph into the other. The operations include node and edge addition and deletion. Calculating the Graph Edit Distance is known to be NP hard. However, as the GED has many applications in network analysis and bioinformatics, many approximations exist.

We decided to base our approach on an algorithm (detailed in figure 3) by Pep Santacruz and Francesc Serratosa [44] for calculating an approximation of GED.

Algorithm 3 Graph Matching with the new confirmation bias

Require: Graph G
Require: Graph G'
Require: [Node, Node] [entry, entry']
1: $pending \leftarrow \{\}$
2: $matching \leftarrow \{\}$
3: $computed \leftarrow \{\}$
4: $S \leftarrow Star(G, entry)$
5: $S' \leftarrow Star(G', entry')$
6: $(D, f) \leftarrow MatchStar(S, S', matching)$
7: Insert $[entry, entry']$ into $computed$
8: Insert $\{[entry, entry'], D, f\}$ into $pending$
9: **while** $pending$ not empty **do**
10: $\{[v, v'], D, f\} \leftarrow MinDistance(pending)$
11: Insert $[v, v']$ into $matching$
12: Delete $\{[v, \sim], \sim, \sim\}$ and $\{[\sim, v'], \sim, \sim\}$ from $pending$
13: $PropagateConfirmationBias(G, G', pending, matching, [v, v'])$
14: **for** each mapping $[w, w']$ such that $w = f(w')$ **do**
15: **if** $[w, w']$ not in $computed$ **then**
16: **if** not $[w, \sim]$ or $[\sim, w']$ in $matching$ **then**
17: $S \leftarrow Star(G, w)$
18: $S' \leftarrow Star(G', w')$
19: $(D, f) \leftarrow MatchStar(S, S', matching)$
20: Insert $[w, w']$ into $computed$
21: Insert $\{[w, w'], D, f\}$ into $pending$
22: **end if**
23: **end if**
24: **end for**
25: **end while**
26: **return** $matching$

[22] In AMD64, the `REP` instruction prefix repeats other instructions, thus creating loops at the instruction level.

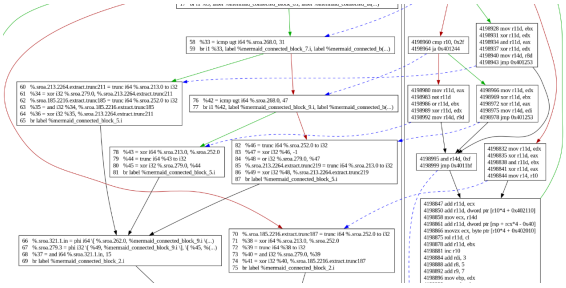
7.4.9.1 The algorithm This algorithm attempts to match nodes in small neighborhoods (the *star* of a node). We start with a seed (in our case the entry point) and then compare neighborhoods finding the best matching possible. These matchings are added to a queue along with the distance between each node. We then iterate on the queue, choosing the matching with the smallest distance and matching its neighborhood.

7.4.9.2 Distance This algorithm relies heavily on a distance between nodes. As we want to compare CFGs written in different languages, the best indicator of distance that we found was the block types (that we defined previously). We created a matrix of transition weights for transforming one type into another. This heuristic is available in table 2. For the *MatchStar* function, we use the Hungarian algorithm with these distances.

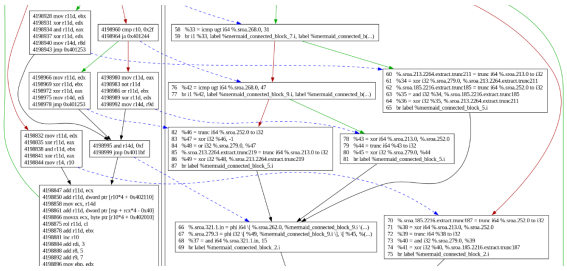
B_1 \ B_2	*Standard*	*Jump*	*Indirect*	*Branch*	*Return*
Standard	1	1.2	1.5	1.5	1.8
Jump		1	1.5	1.5	1.8
Indirect			$\|\|B_1.children\| - \|B_2.children\|\| + 1$	1.2 if $\|B_1.children\| = 2$ else 1.5	1.8
Branch				1	1.8
Return					1

Table 2: Heuristic used to determine the distance between block types
(*IndirectJump* and *IndirectBranch* are treated as the same type called *Indirect*)

7.4.9.3 Limitations Unfortunately, as is, the algorithm did not provide satisfying results for CFGs. For example, let's compare the AMD64 CFG of the MD5 algorithm with its lifted counterpart.[23] If we have a look at figure 40a, we notice a block with three parents on each side. We would want these two blocks to be matched (even more so because each of their three parents are matched up)



(a) CFG matching with original algorithm
(LLVM on the left and AMD64 on the right)



(b) CFG matching with modified algorithm
(AMD64 on the left and LLVM on the right)

Figure 40: Cropped CFG comparison results, matches between blocks are depicted as dotted blue lines

7.4.9.4 Modifications to the algorithm We introduced a new concept to the graph matching algorithm: confirmation bias. We want our matching to be coherent with previous decisions (for example matching the blocks whose parents have been matched together). To do so, our distance now depends on previous matches: two nodes that have been matched (so are in the *matching* set) have a distance of 0. Whenever we add a match, we recalculate every distance to take this into account. This corresponds to the call to $PropagateConfirmationBias$ line 13 of 3. The function is detailed in algorithm 4.

For example, in figure 40 without our modifications, the distance between the nodes with three parents is of at least five (one for self, one for each parent and one for the child). After taking into account the fact

[23] We chose the MD5 hashing function as we needed a reasonably complicated CFG, but that still fit in this report.

that all the parents have previously been matched, the distance can now be decreased to two (one for self, zero for each parent and one for the child).

This modified algorithm provided excellent matching, as seen in figure 40b. This graph comparison was also performed on the obfuscated CFG.

Algorithm 4 Confirmation bias

Require: Graph G
Require: Graph G'
Require: *pending*
Require: *matching*
Require: [Node, Node] $[v, v']$
1: **for** each set $\{[w, w], D, f\}$ in *pending* **do**
2: **if** $\exists\,[x, x'],\ x = f(x')$ and $x = w \wedge x' = w'$ **then**
3: $S \leftarrow Star(G, w)$
4: $S' \leftarrow Star(G', w')$
5: $(D, f) \leftarrow MatchStar(S, S', matching)$
6: **end if**
7: **end for**

7.4.9.5 Terminal output This algorithm provides us with a graph to visualize the differences, but can also provide a textual summary of the differences (as seen in figure 41) and a numerical distance. This is particular useful in a testing pipeline.[24]

```
======= COMPARISON SUMMARY =======
DISTANCE: 2.4

[+] 2 blocks were created (badness: 2)
[+] Created a jump (badness: 0.2)
[+] Created a jump (badness: 0.2)
```

Figure 41: Textual output from CFG comparison

7.4.9.6 Debug symbols We could have also imagined adding debug symbols to our lifted module, when Remill lifts our blocks individually. However, this could be messy after optimization (as seen in Godbolt in figure 42). Furthermore, this would only work with Remill lifted CFGs. We plan on using this algorithm to compare CFGs at different stages of the project, not only during lifting.

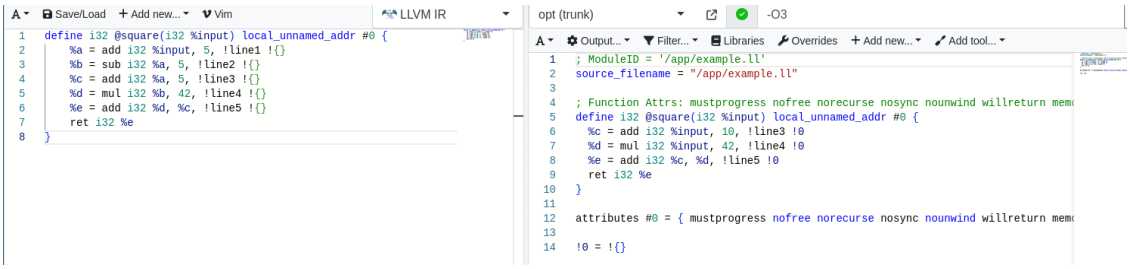


Figure 42: LLVM debug symbols getting lost during optimization

[24] Pipeline we didn't implement.

7.5 Taint analysis

So far, we have extracted a function's CFG from an AMD64 binary and lifted that AMD64 CFG to an LLVM one. We now need to run a taint analysis on our program in order to reconstruct the original (prior to obfuscation) CFG.

This analysis relies heavily on the accuracy of our taint analysis. To ensure our taint analysis is as accurate as possible, we decided to use dynamic taint analysis, as suggested in the TRITON paper.

The only tools that we found, providing dynamic taint analysis in LLVM, were similar to `DataFlowSanitizer` and required the source code. As we only have LLVM bytecode, we decided to write our own rudimentary dynamic taint engine.

7.5.1 The interpreter

Writing a taint engine is not a small feat. To do so in the limited timeframe of this internship, we decided to use KLEE as an inspiration. Indeed, the idea was that a dynamic taint engine is just a very simplified symbolic executor.

After analyzing KLEE's source code, here are a few important takeaways that they use to speed up interpretation:

- KLEE builds a large table to store the value of each variable. This table and the variable IDs are computed prior to the interpretation.
- In this "pre execution phase", KLEE also computes all the constant values. LLVM constants can be quite expensive to compute: `GetElementPointer` constant expression, for example.

Doing all this work prior to the interpretation speeds up the runtime and simplifies writing the interpreter.

7.5.2 Taint precision

Our taint engine only taints data at the variable scale (rather than on the bit scale). This means that if `var1` is tainted and we have `var2` $\leftarrow$ `var1` $\times$ `2`, all of `var2` will be tainted (although its least significant bit should not be).

Yadegari *et al.* [57] argue that taint analysis at the bit-level was needed during their forward taint propagation. We achieved good results with TIGRESS without this level of precision an therefore decided to keep the simpler variable-level taint.

7.5.3 Limitations

The taint engine we built is still very limited and only handles a single datatype: integers. These integers, signed or unsigned, must be encoded on 64 bits or fewer.[25]

Our implementation does not support most of LLVM's other first class types : floating points, metadata, vectors, etc. However, limited support for vectors can be achieved by using the `scalarizer` pass to remove them.[26]

Moreover, instructions were only added to the interpreter if they were needed during the interpretation of one of our test programs. As such, many instructions and constants are not supported.

7.5.4 Results

Leveraging KLEE's strategies, as well as using machine integers rather than Multiple Precision Arithmetic, make the taint engine quite fast.

For comparison, the native LLVM interpreter (`lli`) runs the obfuscated program in around **20ms**, our taint engine runs it in around **10ms**, while native execution takes **under 1ms**. We can explain the speed-up

[25] LLVM allows bit sizes 1 to 2^{24}.

[26] This requires the use of `-scalarize-load-store=true` option to remove vectors in memory operations.

compared to `lli` to come from our limited support for the language and our memory expensive pre-execution passes.

Once again, rather than precise execution times, the takeaway is that this step is fast. To more accurately measure performance, we would need both a larger set of programs and much longer programs.

We have no real way of testing the accuracy of our taint engine. We visually compared our results to those of Triton, and they seemed similar.

Here is the taint analysis result on our lifted CFG.

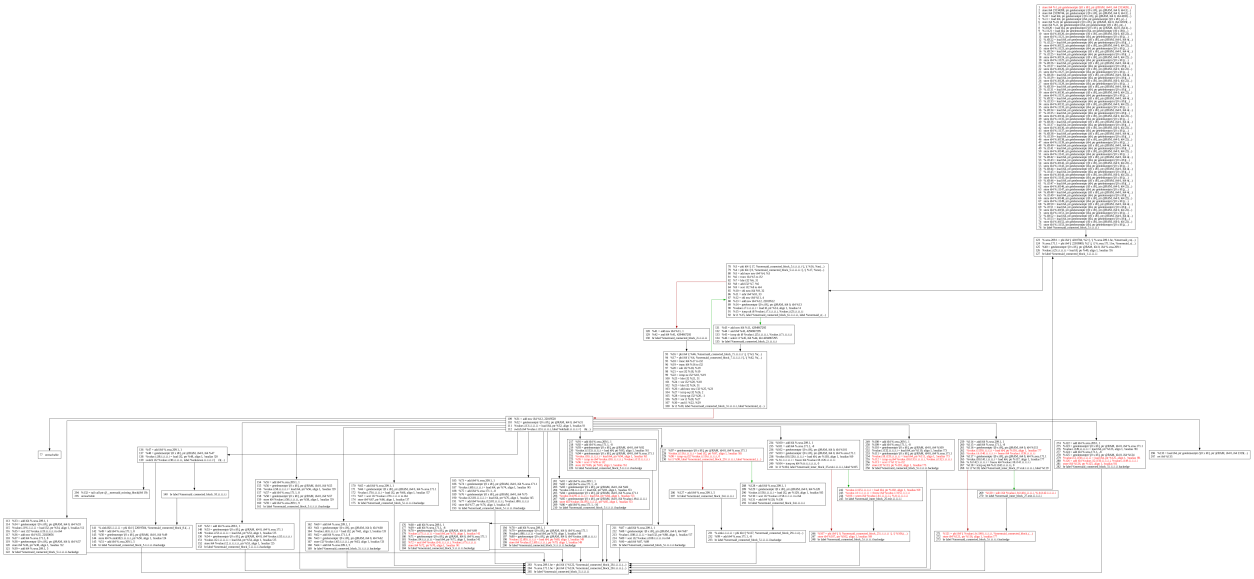


Figure 43: Tainted LLVM CFG of the Collatz function

As expected, only the inside of certain handlers are tainted and only one branching instruction is tainted. We can therefore assume that this instruction is the one used to modify the control flow within the original program.

7.6 Reconstructing a CFG

The next step consists in rebuilding the original CFG using the tainted trace.

During our dynamic taint analysis, we not only construct a tainted CFG, we also keep a record of every block we visited. This record then serves as our trace (removing the need for yet another execution).

7.6.1 Strategy

While TRITON relies on symbolic execution, and Yadegari *et al.* [57] use complex heuristics, we used a simplified (but less resilient) approach.

7.6.1.1 Idea We assimilate untainted control flow instructions (control flow instructions that do not depend on the user input) to opaque predicates. Indeed, tainted control flow instructions are the only places where our trace could change[27]. We therefore create large blocks called *multiblocks*, containing a list of consecutive blocks in our trace and ending with a tainted CFG instruction.

7.6.1.2 Single CFG instruction hypothesis Our strategy relies on perfect taint analysis. However, as our taint engine does not detect indirect data flow one could think that our strategy would not work. Fortunately, we can take advantage of a different hypothesis with VMs. By design, handlers inside a VM are often reused during execution. We therefore hope that the handlers responsible for branching will be tainted at least once. We can then consider the CFG instruction at that address to be always tainted.

7.6.1.3 Example Let's consider a mock example with a program counting up to the user input. Figure 44, represents our initial program.

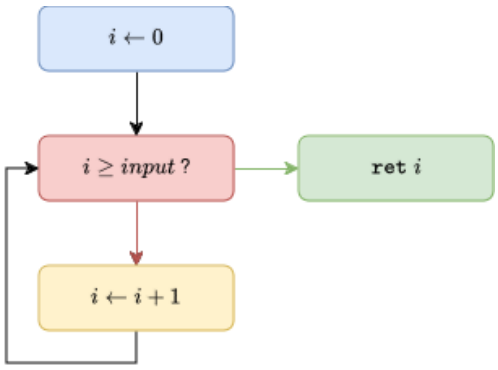


Figure 44: Example counting program

After virtualization, our program might look like figure 45. The VM machinery is depicted in purple: it contains both modifications of the Virtual Program Counter (VPC) (*vpc*) and the dispatcher (block 1).

[27] In reality, it is more complicated than that, we will discuss the details in paragraph 7.6.1.4

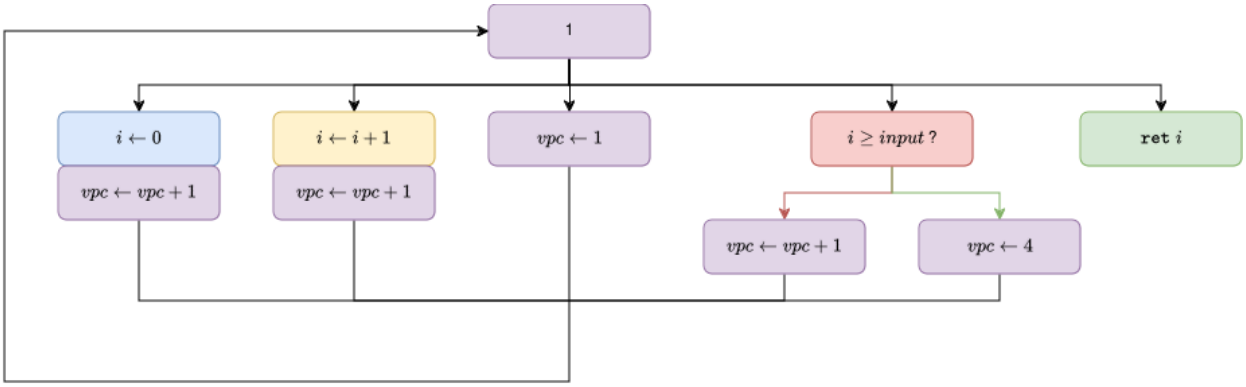


Figure 45: Counting program after virtualization

After our taint analysis, we are able to build a trace of our execution. Figure 46, depicts our trace for $input = 2$. Our taint analysis is also able to taint the CFG instructions in red which contain the *input*.

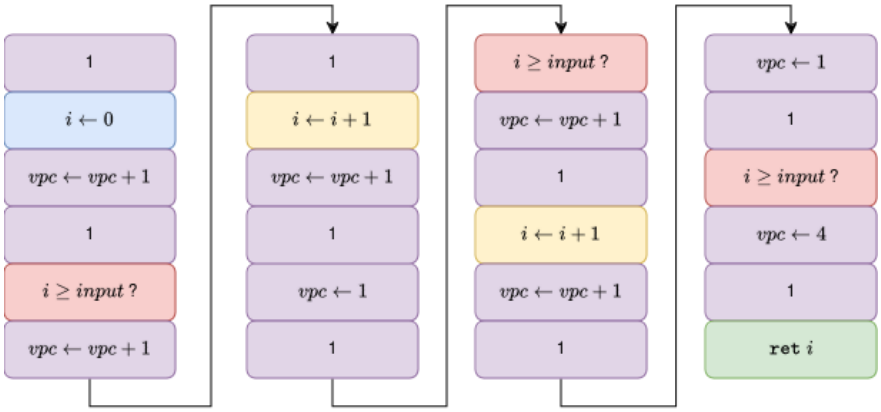


Figure 46: Obfuscated counting program's trace for $input = 2$

We are now able to rebuild our CFG using the algorithm described above: we merge blocks until we reach a tainted CFG instruction (in our example the red block containing "$i \geq input$?"). The resulting CFG contains *multiblocks*: a basic block composed of multiple basic blocks. During construction of our CFG, we merge *multiblocks* containing the same basic blocks (in our case the *multiblock* containing the yellow instruction). Figure 47 contains our reconstructed CFG.

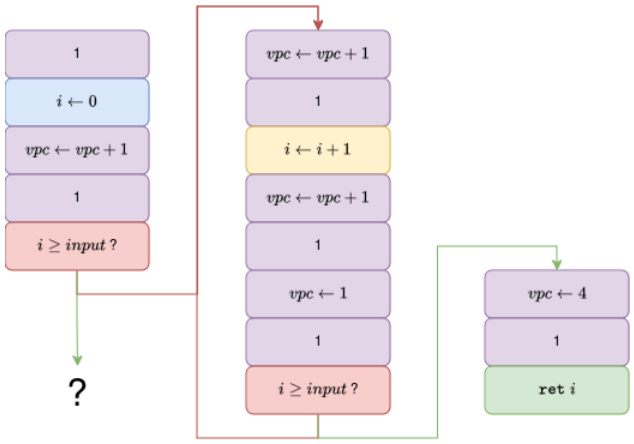


Figure 47: Reconstructed partial CFG

Without any further heuristics, our algorithm produces a slightly oversized CFG containing duplicate blocks (in our example the red CFG instruction).

Furthermore, as we can only split blocks on branch instructions, a lot of information we would get from jumps is lost[28]. In our figure, the jumps are the instructions "$vpx \leftarrow 1$" and "$vpx \leftarrow 4$".

We came up with a simple heuristic to simplify our CFG: *If all the parents of a block have the same postfix, we extract that postfix into a new block.*

The resulting CFG looks strikingly similar to the original CFG, but with additional VM instructions (in purple).

7.6.1.4 Limitations What this algorithm wins in simplicity, it looses in correctness. With our algorithm, we would not be able to differentiate the (pathological) CFGs in figure 48.

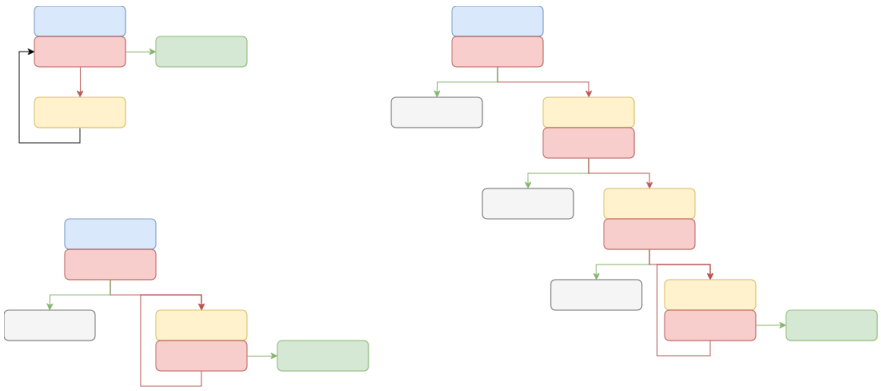


Figure 48: Indistinguashable CFGs to our algorithm

The examples detailed in figure 48 are programs with what would appear in a trace as a loop. To distinguish these programs, we would need to be able to identify the Virtual Program Counter (VPC) (a hard problem).

However, although our algorithm is not always correct, it provides good results most of the time and worked for all the tests we ran during the internship.

[28] We cannot taint an unconditional jump

7.6.2 Implementation

In our tainted trace, we start by removing every untainted CFG instruction (in LLVM these are called `Terminator instructions`). We then rebuild a CFG using only the tainted CFG instructions and a very similar algorithm than that used to build a CFG from a trace (Algorithm 1).

However, our addresses can now belong to multiple *multiblocks* (a basic block can appear in multiple *multiblocks*). To compare *multiblocks*, we therefore can't compare only end addresses.

Instead, we check if *multiblocks* are included in one another by comparing all their constituent basic blocks. As discussed in paragraph 7.6.1.4, this approach does not guarantee correctness.

Using our implementation we were able to rebuild the original CFG of our tainted Collatz function (figure 49)[29].

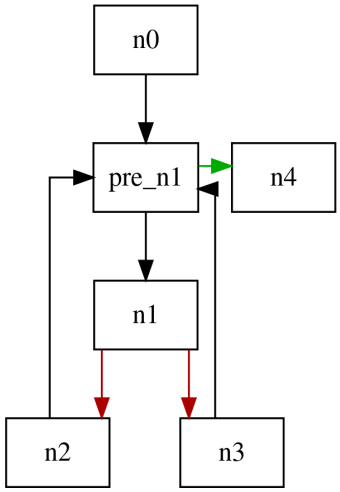


Figure 49: Reconstructed CFG of our obfuscated Collatz function

[29]Note the missing block in this CFG. It is not too big of an issue as LLVM simplifies it, but it is something to keep in mind. Also, the `pre_n1` block is the block constructed from our heuristic.

7.7 From CFG to LLVM module

At this stage, our CFG is not actually code, but simply a CFG containing basic block IDs. In most languages, converting this representation to an executable program would not be too difficult. During a prior experiment in AMD64, we created the program simply by copying each block and pasting them one after another. In LLVM IR things are not that simple.

7.7.1 The challenge with SSA

Indeed, in an SSA language like LLVM IR it is not possible to simply duplicate code.

Let's consider what would happen if we wanted to unroll a loop in a naive way. In figure 50a, we have a simple LLVM loop. The program multiplies the input value by itself for 2 iterations (the result is $pow(input, 4)$).

```
start:
  br label %loop

loop: ; preds = %start, %loop
  %i = phi i32 [ 0, %start ], [ %i_plus_1, %loop ]
  %x = phi i32 [ %input, %start ], [ %x_x, %loop ]
  %x_x = mul nsw i32 %x, %x
  %i_plus_1 = add nuw nsw i32 %i, 1
  %i_eq_2 = icmp eq i32 %i_plus_1, 2
  br i1 %i_eq_2, label %end, label %loop

end:  ; preds = %loop
  ret i32 %x_x
```

(a) An LLVM loop with two iterations

```
start:
  br label %loop0

loop0:
  %i = phi i32 [ 0, %start ], [ %i_plus_1, %loop ]
  %x = phi i32 [ %input, %start ], [ %x_x, %loop ]
  %x_x = mul nsw i32 %x, %x
  %i_plus_1 = add nuw nsw i32 %i, 1
  %i_eq_2 = icmp eq i32 %i_plus_1, 2
  br label %loop1

loop1:
  %i = phi i32 [ 0, %start ], [ %i_plus_1, %loop ]
  %x = phi i32 [ %input, %start ], [ %x_x, %loop ]
  %x_x = mul nsw i32 %x, %x
  %i_plus_1 = add nuw nsw i32 %i, 1
  %i_eq_2 = icmp eq i32 %i_plus_1, 2
  br label %end

end:
  ret i32 %x_x
```

(b) A syntactically incorrect LLVM unrolled loop

If we decide to unroll the loop the naive way: simply by copying and pasting the blocks as we did in AMD64, the result is the program in figure 50b. The program in figure 50b is not correct LLVM IR. Indeed, values (both integers and labels) are being redefined — which is not possible in an SSA language.

We therefore need to find a way to "fix" this program.

7.7.2 The "algorithm"

This approach was inspired by the code used for loop unrolling in LLVM. To solve the incorrect program in figure 50b, we need an LLVM-pass that remaps values. This is done in two passes.

7.7.2.1 Remapping *interior values* The first pass is done inside each block. We will be using a map to assign a "new value" to "old" ones. To do so, we visit every expression adding the destination to the map and attempting to replace operands. If an operand is not in our map, we add it to a list of unmapped values. We initialize our map with the labels of the children.

For example, let's consider the first block of the unrolled loop in figure 50b, this block becomes `%loop0` in figure 51. At first our map contains only the labels of the children (`%loop` → `%loop1`). The first two instructions of our loop are phi nodes. In this pass, as we are only interested in interior values, most of the instruction is ignored. However, we can still add the destinations to our map: we will therefore add `%i` → `%l0_i` and `%x` → `%l0_x`. When we reach the third instruction (`mul`), our map contains both operands, we can therefore replace the operands of our instruction with the new values: the instruction becomes `mul nsw i32 %l0_x, %l0_x`. And once again, we will add the destination to the map (`%x_x` → `%l0_x_x`). By proceeding this way, we are able to create the IR of figure 51.

```
start:
  br label %loop0

loop0:                          ; preds = %start
  ; BROKEN
  %l0_i = phi i32 [ 0, %start], [%i_plus_1, %loop ]
  ; BROKEN
  %l0_x = phi i32 [ %input, %start ], [ %x_x, %loop0 ]

  %l0_x_x = mul nsw i32 %l0_x, %l0_x
  %l0_i_plus_1 = add nuw nsw i32 %l0_i, 1
  %l0_i_eq_2 = icmp eq i32 %l0_i_plus_1, 3
  br label %loop1

loop1:                          ; preds = %loop0
  ; BROKEN
  %l1_i = phi i32 [ 0, %start], [%i_plus_1, %loop ]
  ; BROKEN
  %l1_x = phi i32 [ %input, %start ], [ %x_x, %loop0 ]

  %l1_x_x = mul nsw i32 %l1_x, %l1_x
  %l1_i_plus_1 = add nuw nsw i32 %l1_i, 1
  %l1_i_eq_2 = icmp eq i32 %l1_i_plus_1, 2
  br label %end

end:                                ; preds = %loop1
  ; BROKEN
  ret i32 %x_x
```

Figure 51: Loop after fist remapping pass

7.7.2.2 Remapping *exterior values*

The previous pass is very effective for "fixing" the end of blocks, however some values (especially at the beginning of blocks) might remain unmapped. The second pass therefore fixed initial values and phi nodes.

In both cases, we will look for a value in the parent's map. For example, in the `%end` block, the value `%x_x` was unmapped. For each parent of `%end` (in this case `%loop1`), we will look for the mapped value (in this case, the map of `%loop1` contains `%x_x` $\rightarrow$ `%l1_x_x` so our mapped value is `%l1_x_x`) and create a phi node matching each block with its mapped value (in this case the phi node would be `phi i32 [ %l1_x_x, %loop1 ]`).[30]

```
start:
  br label %loop0

loop0:                            ; preds = %start
  %l0_i = phi i32 [ 0, %start ]
  %l0_x = phi i32 [ %input, %start ]
  %l0_x_x = mul nsw i32 %l0_x, %l0_x
  %l0_i_plus_1 = add nuw nsw i32 %l0_i, 1
  %l0_i_eq_2 = icmp eq i32 %l0_i_plus_1, 3
  br label %loop1

loop1:                            ; preds = %loop0
  %l1_i = phi i32 [ %l0_i_plus_1, %loop0 ]
  %l1_x = phi i32 [ %l0_x_x, %loop0 ]
  %l1_x_x = mul nsw i32 %l1_x, %l1_x
  %l1_i_plus_1 = add nuw nsw i32 %l1_i, 1
  %l1_i_eq_2 = icmp eq i32 %l1_i_plus_1, 2
  br label %end

end:                              ; preds = %loop1
  %end_x_x = phi i32 [ %l1_x_x, %loop1 ]
  ret i32 %end_x_x
```

Figure 52: Loop after both remapping passes

7.7.3 Results

The result of both passes can be seen in figure 52. These passes were surprising long in our experiments, taking roughly as long as the taint analysis (10ms).

We were able to test that the resulting program was correct by interpreting it and comparing its input output behavior with the original program.

[30] Adding a phi node to a block with a single parent may seem unnecessary, but it makes these passes a lot easier to write. LLVM's optimizations will take care of removing this superfluous phi.

8 LLVM optimizations

Our deobfuscation pipeline makes use of LLVM's optimizations to simplify the program. During this internship, we briefly experimented with creating dedicated pipelines and new passes. We observed that `-O3`, `-Os` and `-Oz` all behave similarly on our samples. Indeed, we are mainly interested in the constant folding and constant propagation passes. We created a simple pipeline `-Om`, inspired by `-O3` but with a loop to run certain passes multiple times (see figure 53). We simply run these passes five times whereas in Saturn passes are apparently applied until a fix point is reached.

```
for (size_t I = 0; I < 5; I++) {
  FPM.addPass(ConstantConcretizationPass());
  FPM.addPass(GVNPass());
  FPM.addPass(SCCPPass());
  FPM.addPass(BDCEPass());
  FPM.addPass(MemoryCoalescePass());
  FPM.addPass(DSEPass());
  FPM.addPass(InstCombinePass());
}
```

Figure 53: An extract of the `Om` LLVM pipeline

This pipeline also contains custom passes: `ConstantConcretizationPass` and `MemoryCoalescePass`. We reimplemented these passes that were designed for Saturn. The `ConstantConcretizationPass` attempts to read constants from the binary (such as the virtualized program or the jump table). The `MemoryCoalescePass` is a more complex pass that attempts to concretize a `load` that follows overlapping `store`s.

9 Results

Running our deobfuscation pipeline from end to end takes **under half a second**. This is quicker than running the programs individually as we are able to skip some serializing/deserializing. However, we believe a large part of that time is spent reading and writing to files, as not all programs communicate directly.

Although we obtain results very quickly, they remain imperfect for the moment. As it stands, loops and pointers of pointers are not properly deobfuscated.

9.1 Loop invariants

Although LLVM greatly simplified our code (our obfuscated module went from 13K instructions to 400), the code is still not readable. After careful inspection, one instruction was responsible for this "clog" (figure 54).

```
%4 = phi i64 [ %.be, %new-1.backedge ], [ 4210868, %new-0 ]
```

Figure 54: Single instruction preventing deobfuscation

Here `%4` is the Virtual Program Counter (VPC) at the beginning of the loop, `4210868` is the value of the VPC on the loop's first iteration and `%.be` is the value of the VPC after an iteration. The issue is that LLVM is unable to verify that this is an invariant.

In other words, LLVM can't prove that after going around a loop we arrive at the same position.

This issue completely prevents LLVM from simplifying the code as without the VPC it is unable to fetch the instructions or their immediates.

We believe that this is an issue TRITON must not have faced as symbolic execution should be able to prove that `%4` is invariant.

9.2 After a nudge

We manually fixed the loop invariant (by replacing `%.be` with `4210868` in the phi node) to see what would happen. The results are very promising. Figure 55 shows an extract of the deobfuscated code. In this extract, we can clearly see the logic behind the Collatz function: we have a parity check and then either divide by two or multiply by three and add one.

```
new-1:                                            ; preds = %new-0, %new-1
  %3 = phi i64 [ %11, %new-1 ], [ %2, %new-0 ]
  %4 = and i64 %3, 1
  %5 = icmp eq i64 %4, 0
  %6 = mul i64 %3, 3
  %7 = add i64 %6, 1
  %8 = lshr exact i64 %3, 1
  %.sink = select i1 %5, i64 %8, i64 %7
```

Figure 55: An extract of the deobfuscated Collatz function

9.2.0.1 Noise The resulting code is however not perfect, it is quite noisy (the total module is around 100 lines long). We identified two main reasons:

- **Dead writes to `@RAM`**. The resulting program contains at least 40 unused writes to the `@RAM` global. LLVM is unable to identify these as dead since `@RAM` is an external global and could therefore be read elsewhere. We believe a simple pass could solve this issue.
- **Pointers of pointers**. A more difficult challenge is the presence of pointers of pointers. These prevent LLVM's antialiasing analysis from being effective.

9.3 Going back to our initial program

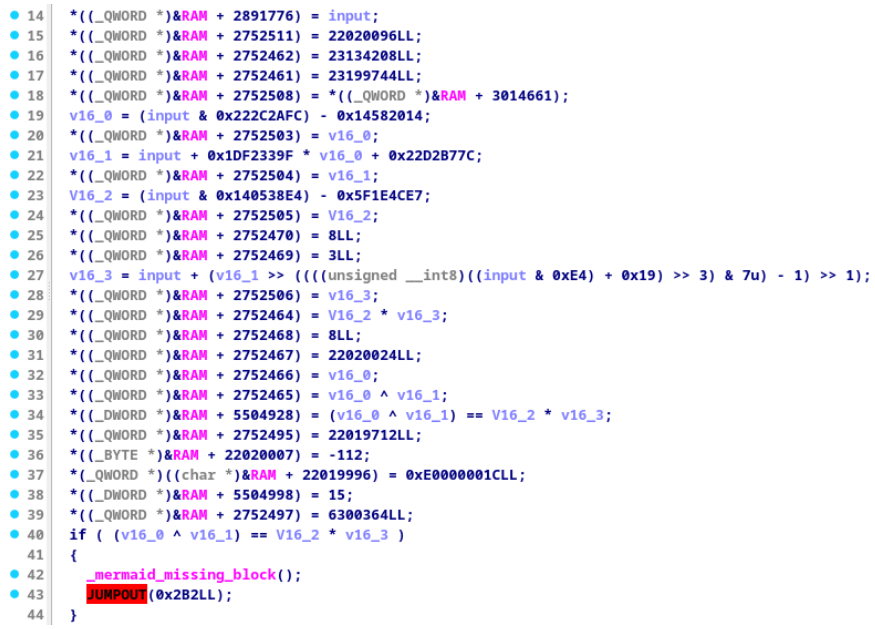
For the sake of storytelling, we decided to run our deobfuscator on the TIGRESS challenge we manually deobfuscated at the beginning of the internship.

There were a couple final struggles when attempting to run our deobfuscator on the challenges: different compiler options, new instructions etc.

After some work we were able to run the deobfuscation on our original obfuscated program: `challenge0-1`. This program does not contain any loops so our deobfuscator did not have any difficulties.

Our script once again ran in under a second and yielded a partial CFG (available in appendix A.4). The CFG is partial since this program has two execution paths. We were therefore only able to construct half of the CFG.

To facilitate the comparison, we compiled the deobfuscated LLVM module and opened the binary in IDA. Figure 56 shows a decompiled extract, and figure 57 shows the same code manually cleaned up (without the unused `RAM` writes).



```
14  *((_QWORD *)&RAM + 2891776) = input;
15  *((_QWORD *)&RAM + 2752511) = 22020096LL;
16  *((_QWORD *)&RAM + 2752462) = 23134208LL;
17  *((_QWORD *)&RAM + 2752461) = 23199744LL;
18  *((_QWORD *)&RAM + 2752508) = *((_QWORD *)&RAM + 3014661);
19  v16_0 = (input & 0x222C2AFC) - 0x14582014;
20  *((_QWORD *)&RAM + 2752503) = v16_0;
21  v16_1 = input + 0x1DF2339F * v16_0 + 0x22D2B77C;
22  *((_QWORD *)&RAM + 2752504) = v16_1;
23  V16_2 = (input & 0x140538E4) - 0x5F1E4CE7;
24  *((_QWORD *)&RAM + 2752505) = V16_2;
25  *((_QWORD *)&RAM + 2752470) = 8LL;
26  *((_QWORD *)&RAM + 2752469) = 3LL;
27  v16_3 = input + (v16_1 >> ((((unsigned __int8)((input & 0xE4) + 0x19) >> 3) & 7u) - 1) >> 1);
28  *((_QWORD *)&RAM + 2752506) = v16_3;
29  *((_QWORD *)&RAM + 2752464) = V16_2 * v16_3;
30  *((_QWORD *)&RAM + 2752468) = 8LL;
31  *((_QWORD *)&RAM + 2752467) = 22020024LL;
32  *((_QWORD *)&RAM + 2752466) = v16_0;
33  *((_QWORD *)&RAM + 2752465) = v16_0 ^ v16_1;
34  *((_DWORD *)&RAM + 5504928) = (v16_0 ^ v16_1) == V16_2 * v16_3;
35  *((_QWORD *)&RAM + 2752495) = 22019712LL;
36  *((_BYTE *)&RAM + 22020007) = -112;
37  *(_QWORD *)((char *)&RAM + 22019996) = 0xE0000001CLL;
38  *((_DWORD *)&RAM + 5504998) = 15;
39  *((_QWORD *)&RAM + 2752497) = 6300364LL;
40  if ( (v16_0 ^ v16_1) == V16_2 * v16_3 )
41  {
42    _mermaid_missing_block();
43    JUMPOUT(0x2B2LL);
44  }
```

Figure 56: Decompiled extract of the `challenge01` program deobfuscated with Mermaid

```
v16_0 = (input & 0x222C2AFC) - 0x14582014;
v16_1 = input + 0x1DF2339F * v16_0 + 0x22D2B77C;
V16_2 = (input & 0x140538E4) - 0x5F1E4CE7;
v16_3 =
    input + (v16_1 >>
             ((((unsigned __int8)((input & 0xE4) + 0x19) >> 3) & 7u) - 1) >> 1);
if ((v16_0 ^ v16_1) == V16_2 * v16_3) {
  _mermaid_missing_block();
  JUMPOUT(0x2B2LL);
}
```

Figure 57: Decompiled extract of the `challenge01` program without `RAM` writes

We can compare the automatic deobfuscation extract from figure 57 with the manually deobfuscated code from figure 26 and observe that the constants and operations are identical.

The rest of the deobfuscated program is given in appendix B.1, we believe it is well deobfuscated — although still noisy.

Conclusion

During this internship, we explored the vast topic of deobfuscation and in particular the challenges faced by automatic deobfuscators. We conducted a state of the art, experimented with state of the art tools and developed our own tool.

The state of the art established that virtualization obfuscation was currently one of the best obfuscation techniques. We presented how different deobfuscation tools deal with it. We also focused on other obfuscation and deobfuscation strategies, to provide a comprehensive introduction to the field of deobfuscation.

This internship allowed us to create a new reverse engineering tool leveraging LLVM: Mermaid. This tool is able to construct, display and save CFGs, lift binaries and perform dynamic taint analysis in LLVM. We were able to use Mermaid to partially deobfuscate simple binaries obfuscated with Tigress. This deobfuscation step was very fast when compared to Triton's deobfuscation times.

Mermaid is however still a rather immature tool: many simplifying hypotheses were made, both on the nature of the obfuscated program and the type of obfuscation. Moreover, the results are partial and the tool struggles with loops.

In the future, we would like to extend Mermaid's capabilities by removing some of these hypotheses. We would also like to integrate an abstract interpretation engine, which we believe could solve our invariant issue.

A CFGs

All these CFG were generated with LibCFG.

A.1 x86 CFG of the Collatz function

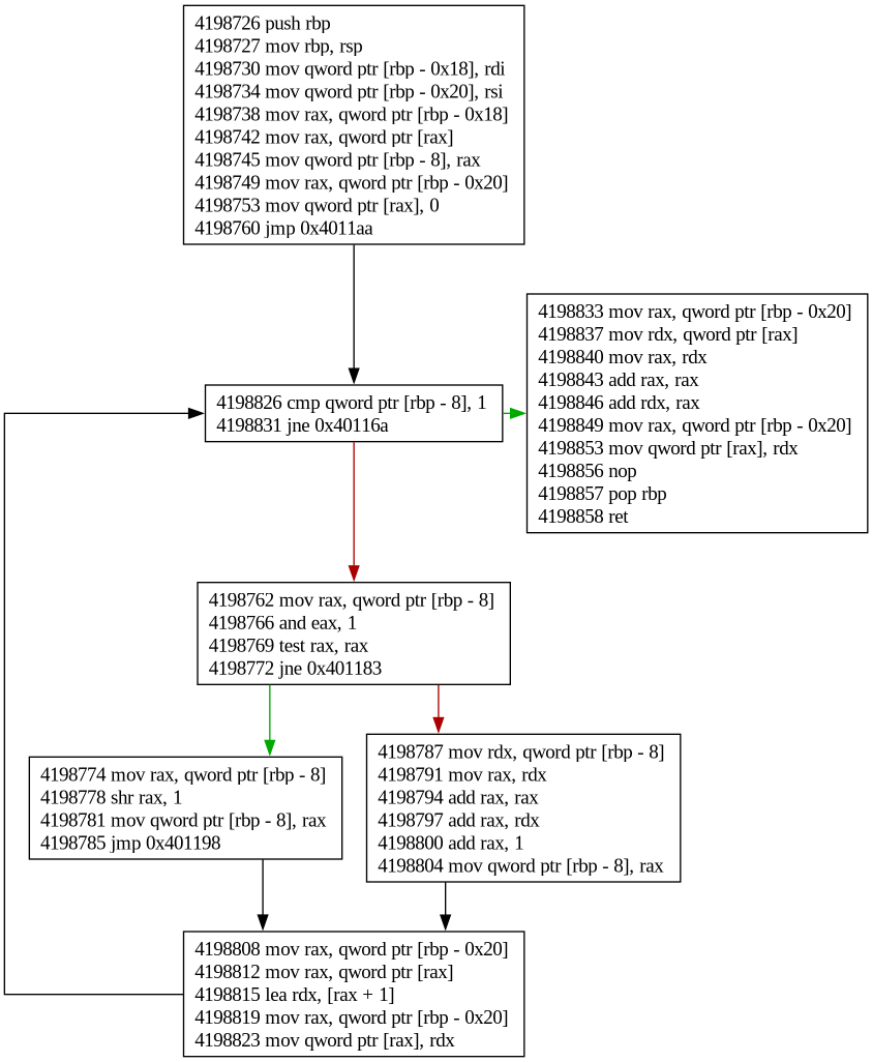


Figure 58: x86 CFG of the Collatz function

A.2 Partial x86 CFG of the obfuscated Collatz function

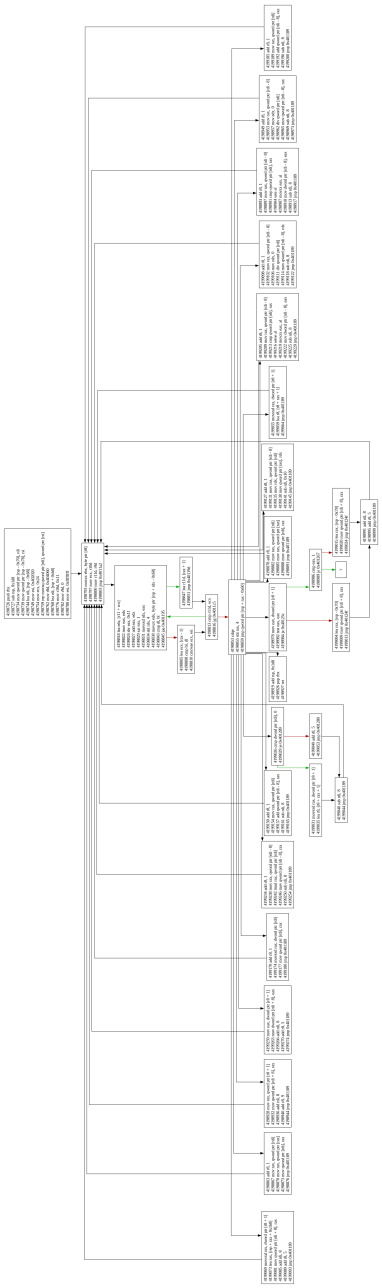


Figure 59: Partial x86 CFG of the obfuscated Collatz function

A.3 Partial lifted CFG of the obfuscated Collatz function

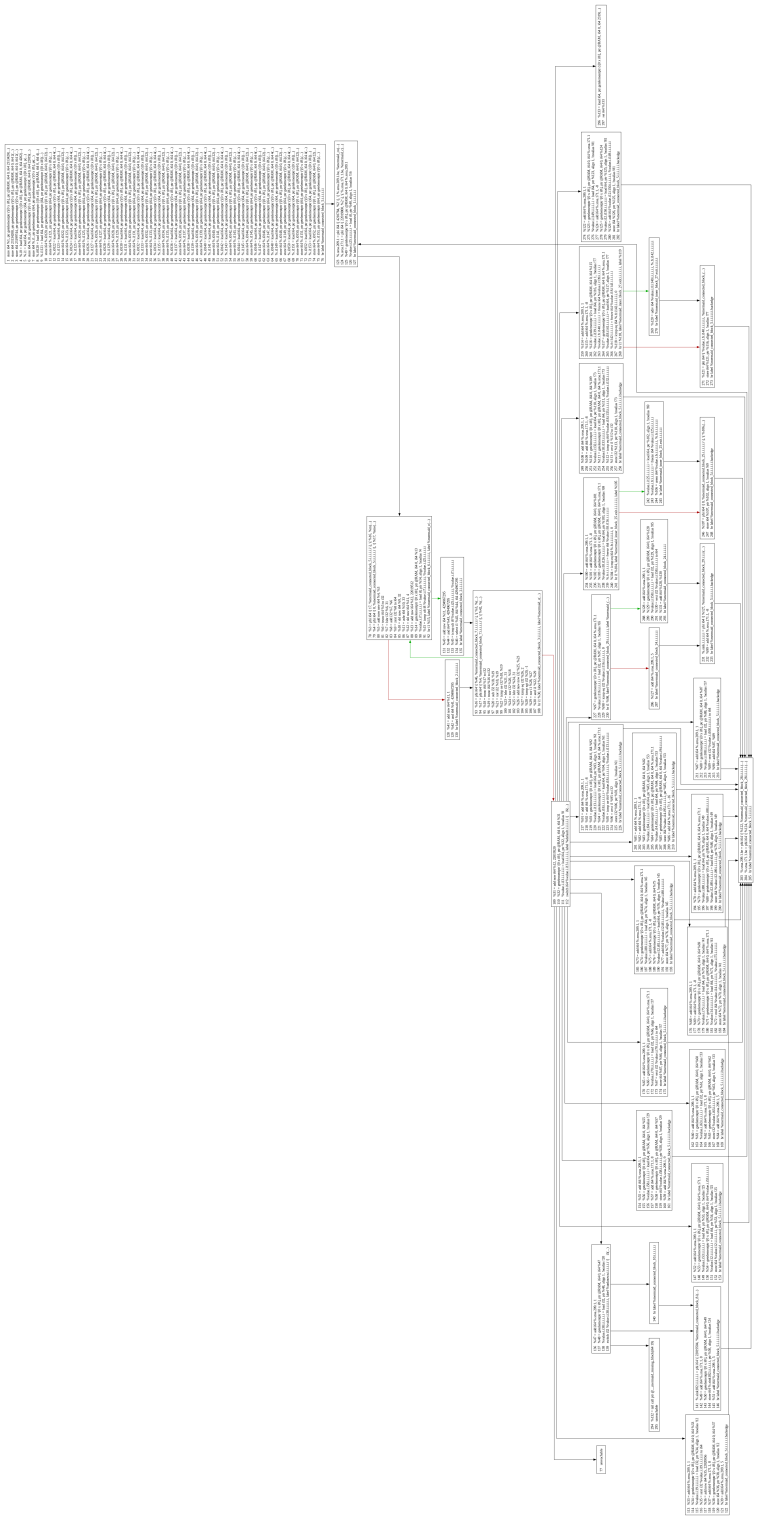


Figure 60: Partial lifted CFG of the obfuscated Collatz function

A.4 Reconstructed CFG of the `challenge01` program deobfuscated with MERMAID

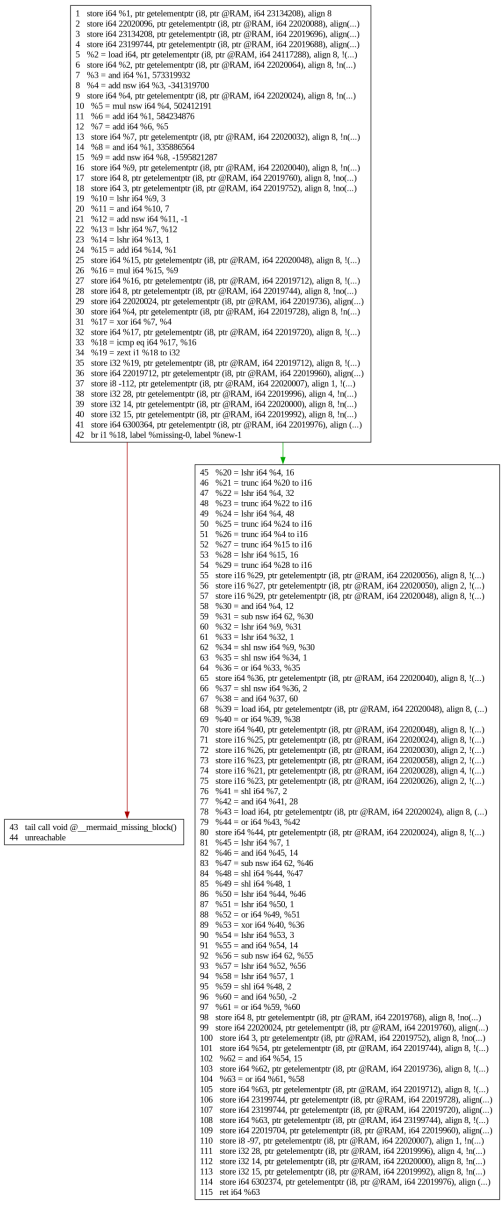


Figure 61: Reconstructed CFG of the `challenge01` program deobfuscated with MERMAID

B Code

B.1 Decompiled code of deobfuscated challenge01 function

```
unsigned __int64 __fastcall mermaid(__int64 out, __int64 input) {
  __int64 v16_0;           // rax
  unsigned __int64 v16_1;  // rdx
  unsigned __int64 V16_2;  // r8
  __int64 v16_3;           // r9
  unsigned __int64 v6;     // r8
  __int64 v7;              // r9
  unsigned __int64 v8;     // rax
  unsigned __int64 v9;     // rax
  unsigned __int64 v10;    // r9
  unsigned __int64 result; // rax

  *((_QWORD *)&RAM + 2891776) = input;
  *((_QWORD *)&RAM + 2752511) = 22020096LL;
  *((_QWORD *)&RAM + 2752462) = 23134208LL;
  *((_QWORD *)&RAM + 2752461) = 23199744LL;
  *((_QWORD *)&RAM + 2752508) = *((_QWORD *)&RAM + 3014661);
  v16_0 = (input & 0x222C2AFC) - 0x14582014;
  *((_QWORD *)&RAM + 2752503) = v16_0;
  v16_1 = input + 0x1DF2339F * v16_0 + 0x22D2B77C;
  *((_QWORD *)&RAM + 2752504) = v16_1;
  V16_2 = (input & 0x140538E4) - 0x5F1E4CE7;
  *((_QWORD *)&RAM + 2752505) = V16_2;
  *((_QWORD *)&RAM + 2752470) = 8LL;
  *((_QWORD *)&RAM + 2752469) = 3LL;
  v16_3 = input +
          (v16_1 >>
           ((((unsigned __int8)((input & 0xE4) + 0x19) >> 3) & 7u) - 1) >> 1);
  *((_QWORD *)&RAM + 2752506) = v16_3;
  *((_QWORD *)&RAM + 2752464) = V16_2 * v16_3;
  *((_QWORD *)&RAM + 2752468) = 8LL;
  *((_QWORD *)&RAM + 2752467) = 22020024LL;
  *((_QWORD *)&RAM + 2752466) = v16_0;
  *((_QWORD *)&RAM + 2752465) = v16_0 ^ v16_1;
  *((_DWORD *)&RAM + 5504928) = (v16_0 ^ v16_1) == V16_2 * v16_3;
  *((_QWORD *)&RAM + 2752495) = 22019712LL;
  *((_BYTE *)&RAM + 22020007) = -112;
  *(_QWORD *)((char *)&RAM + 22019996) = 0xE0000001CLL;
  *((_DWORD *)&RAM + 5504998) = 15;
  *((_QWORD *)&RAM + 2752497) = 6300364LL;
  if ((v16_0 ^ v16_1) == V16_2 * v16_3) {
    _mermaid_missing_block();
    JUMPOUT(0x2B2LL);
  }
  *((_WORD *)&RAM + 11010028) = WORD1(v16_3);
  *((_WORD *)&RAM + 11010025) = v16_3;
  *((_WORD *)&RAM + 11010024) = WORD1(v16_3);
  v6 = ((unsigned int)(V16_2 >> (v16_0 & 0xC ^ 0x3E)) >> 1) | (2 * V16_2);
  *((_QWORD *)&RAM + 2752505) = v6;
  v7 = *((_QWORD *)&RAM + 2752506) | (4 * (_BYTE)v6) & 0x3C;
  *((_QWORD *)&RAM + 2752506) = v7;
  *((_WORD *)&RAM + 11010012) = HIWORD(v16_0);
  *((_WORD *)&RAM + 11010015) = v16_0;
  *((_WORD *)&RAM + 11010029) = WORD2(v16_0);
  *((_WORD *)&RAM + 11010014) = WORD1(v16_0);
  *((_WORD *)&RAM + 11010013) = WORD2(v16_0);
  v8 = *((_QWORD *)&RAM + 2752503) | (4 * (_BYTE)v16_1) & 0x1C;
  *((_QWORD *)&RAM + 2752503) = v8;
  v9 = (2 * (v8 << (((unsigned __int8)v16_1 >> 1) & 0xE ^ 0x3Eu))) |
       (v8 >> (((unsigned __int8)v16_1 >> 1) & 0xE) >> 1);
  v10 = (v6 ^ v7) >> 3;
  *((_QWORD *)&RAM + 2752471) = 8LL;
  *((_QWORD *)&RAM + 2752470) = 22020024LL;
  *((_QWORD *)&RAM + 2752469) = 3LL;
  *((_QWORD *)&RAM + 2752468) = v10;
  *((_QWORD *)&RAM + 2752467) = v10 & 0xF;
  result = ((unsigned int)(v9 >> (v10 & 0xE ^ 0x3E)) >> 1) |
           (2 * (v9 << (v10 & 0xE)));
  *((_QWORD *)&RAM + 2752464) = result;
  *((_QWORD *)&RAM + 2752466) = 23199744LL;
  *((_QWORD *)&RAM + 2752465) = 23199744LL;
  *((_QWORD *)&RAM + 2899968) = result;
  *((_QWORD *)&RAM + 2752495) = 22019704LL;
  *((_BYTE *)&RAM + 22020007) = -97;
  *(_QWORD *)((char *)&RAM + 22019996) = 0xE0000001CLL;
  *((_DWORD *)&RAM + 5504998) = 15;
  *((_QWORD *)&RAM + 2752497) = 6302374LL;
  return result;
}
```

Figure 62: Decompiled code of deobfuscated challenge01 function

References

[1] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An {In-Depth} Analysis of Disassembly on {Full-Scale} x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, 2016 (cited on page 3).

[2] Peter Ferrie. ANTI-UNPACKER TRICKS. `https://pferrie.tripod.com/papers/unpackers.pdf` (cited on page 10).

[3] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs (cited on page 6).

[4] binexport. `https://github.com/google/binexport` (cited on page 29).

[5] Philippe Biondi, Raphaël Rigo, Sarah Zennou, and Xavier Mehrenberger. BinCAT: purrfecting binary static analysis (cited on page 4).

[6] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, 2017 (cited on page 12).

[7] Adam Burgher. BackdoorDiplomacy: Upgrading from Quarian to Turian. `https://www.welivesecurity.com/2021/06/10/backdoordiplomacy-upgrading-quarian-turian/` (cited on page 1).

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (cited on page 5).

[9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013. DOI: `10.1145/2408776.2408795` (cited on pages 4, 5).

[10] 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*, pages 319–328, 2012. DOI: `10.1145/2420950.2420997` (cited on pages 6, 10, 13, 14).

[11] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998. DOI: `10.1145/268946.268962` (cited on page 6).

[12] The MITRE corporation. MITRE ATT&CK. `https://attack.mitre.org/` (cited on page 1).

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977. DOI: `10.1145/512950.512973` (cited on page 4).

[14] Hoang-Vu Dang and Anh-Quynh Nguyen. Unicorn: next generation cpu emulator framework. In 2015 (cited on page 27).

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. DOI: `10.1007/978-3-540-78800-3_24` (cited on page 5).

[16] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017. DOI: `10.1145/3033019.3033028` (cited on pages 3, 4).

[17] Adel Djoudi and Sébastien Bardin. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035, pages 212–217, 2015. DOI: `10.1007/978-3-662-46681-0\_17` (cited on pages 3, 4).

[18] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. CaDeCFF: Compiler-Agnostic Deobfuscator of Control Flow Flattening. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 282–291, 2022. DOI: 10.1145/3545258.3545269 (cited on pages 3, 11).

[19] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 27–38, 2016. DOI: 10.1145/2995306.2995308 (cited on pages 7, 10, 12).

[20] Martin Co Fernando Merces Byron Gelera. KillDisk Variant Hits Latin American Finance Industry. https://www.trendmicro.com/en_us/research/18/f/new-killdisk-variant-hits-latin-american-financial-organizations-again.html (cited on page 1).

[21] Peter Garba and Matteo Favaro. Saturn - software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 27–38, 2019. DOI: 10.1145/3338503.3357721 (cited on pages 11, 33).

[22] Josh Grunzweig. Comnie Continues to Target Organizations in East Asia. https://unit42.paloaltonetworks.com/unit42-comnie-continues-target-organizations-east-asia/ (cited on page 1).

[23] Intel Processor Trace. https://grasland.pages.in2p3.fr/tp-perf/perf-script/intel-pt.html (cited on page 27).

[24] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2R2: Building an Efficient Front-End for Binary Analysis. *Proceedings 2019 Workshop on Binary Analysis Research*, 2019. DOI: 10.14722/bar.2019.23051 (cited on page 3).

[25] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9, 2015. DOI: 10.1109/SPRO.2015.10 (cited on pages 6, 8, 10).

[26] Johannes Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *2012 19th Working Conference on Reverse Engineering*, pages 61–70, 2012. DOI: 10.1109/WCRE.2012.16 (cited on page 12).

[27] 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*, pages 1–15, 2021. DOI: 10.1145/3465481.3465772 (cited on pages 8, 10, 11).

[28] T Laszlo and A Kiss. OBFUSCATING C++ PROGRAMS VIA CONTROL FLOW FLATTENING (cited on page 8).

[29] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In pages 75–88, 2004 (cited on page 3).

[30] Landon Curt Noll Leo Broukhis. The International Obfuscated C Code Contest. https://www.ioccc.org/ (cited on page 1).

[31] McSEMA. https://github.com/lifting-bits/mcsema (cited on page 31).

[32] miasm. https://github.com/cea-sec/miasm (cited on page 3).

[33] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. DOI: 10.1145/1273442.1250746 (cited on page 30).

[34] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965, pages 3–17, 2023. DOI: 10.1007/978-3-031-37703-7_1 (cited on page 5).

[35] Nathan Otterness. Tiny ELF Files: Revisited in 2021. https://nathanotterness.com/2021/10/tiny_elf_modernized.html (cited on page 2).

[36] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. *2021 IEEE Symposium on Security and Privacy (SP)*:833–851, 2021. DOI: 10.1109/SP40001.2021.00012 (cited on pages 2–5).

[37] Quokka. https://github.com/quarkslab/quokka (cited on page 29).

[38] Radare2. https://rada.re/n/ (cited on page 3).

[39] Remill. https://github.com/lifting-bits/remill (cited on pages 11, 31).

[40] H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953 (cited on page 2).

[41] Andrei Rimsa, José Amaral, and Fernando Pereira. Practical dynamic reconstruction of control flow graphs. *Software: Practice and Experience*, 51, 2020. DOI: 10.1002/spe.2907 (cited on page 30).

[42] Jonathan Salwan. Triton. https://github.com/JonathanSalwan/Triton (cited on page 25).

[43] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. *Symbolic deobfuscation: from virtualized code back to the original*. In 2018, pages 372–392. DOI: 10.1007/978-3-319-93411-2_17 (cited on pages 13, 14, 25).

[44] Pep Santacruz and Francesc Serratosa. Error-tolerant graph matching in linear computational cost using an initial small partial matching. *Pattern Recognition Letters*, 134:10–19, 2020. DOI: https://doi.org/10.1016/j.patrec.2018.04.003. Applications of Graph-based Techniques to Pattern Recognition (cited on page 37).

[45] Florent Saudel and Jonathan Salwan. Triton: a dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications*, pages 31–54, 2015 (cited on page 25).

[46] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening Code Obfuscation Against Automated Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, 2022 (cited on pages 3, 4, 6, 7, 9, 10, 12).

[47] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C (cited on page 5).

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. DOI: 10.1109/SP.2016.17 (cited on pages 3–5).

[49] SMT-Comp 2023. https://smt-comp.github.io/2023/ (cited on page 5).

[50] Themida. https://www.oreans.com/Themida.php (cited on page 13).

[51] Tigress Challenge. http://tigress.cs.arizona.edu/challenges.html (cited on pages 14, 22, 24).

[52] Moritz Schloegel Tim Blazytko. The Next Generation of Virtualization based Obfuscators. Recon 2022 conference (cited on page 8).

[53] VMAttack. https://github.com/anatolikalysch/VMAttack (cited on page 10).

[54] vmprotect. https://vmpsoft.com/ (cited on pages 10, 13).

[55] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In *Machine Learning and Knowledge Discovery in Databases*, pages 522–536, 2011. DOI: 10.1007/978-3-642-23808-6_34 (cited on page 2).

[56] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. Vmhunt: a verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 442–458, 2018. DOI: 10.1145/3243734.3243827 (cited on page 10).

[57] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, 2015. DOI: 10.1109/SP.2015.47 (cited on pages 13, 26, 37, 40, 42).

[58] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications*, pages 61–75, 2007 (cited on page 7).

Acronyms

CFG Control Flow Graph. 3, 8, 9, 11, 12, 16, 22, 24, 26–30, 33, 34, 36–46, 50, 52

GEP Get Element Pointer. 18–21

IR Intermediate Representation. 3, 4, 11, 15–17, 26, 29–33, 46, 47

ISA Instruction Set Architecture. 2–4

MBA Mixed Boolean-Arithmetic. 7, 10–12

SMT Satisfiability Modulo Theory. 4, 5, 7, 11

SSA Static Single Assignement. 16–18, 31, 46

VM Virtual Machine. 9, 12, 22

VPC Virtual Program Counter. 12, 23, 42, 44, 49