Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Una técnica de ofuscación basada en una máquina virtual

Tesis de Licenciatura en Ciencias de la Computación

Manuel Carrasco

Director: Dr. Juan Manuel Martinez Caamaño

Codirector: Dr. Diego Garbervetsky

Buenos Aires, 2019

UNA TÉCNICA DE OFUSCACIÓN BASADA EN UNA MÁQUINA VIRTUAL

En este trabajo presentamos una ofuscación basada en una máquina virtual. La máquina virtual es un intérprete de un set arbitrario de instrucciones. El código a ofuscar es traducido a un programa válido para el intérprete. Esto permite ejecutar el intérprete en vez del código original. En caso que un atacante realice ingeniería reversa, idealmente, debería tener que entender todo el funcionamiento del interprete. En esta tesis evaluamos nuestra ofuscación con especialistas en ingenería reversa. Como resultado de la evaluación encontramos que nuestra ofuscación fue resistente a ataques clásicos pero inútil contra herramientas del estado del arte. Esto nos permitió desarrollar dos nuevas ofuscaciones en las que mitigamos las vulnerabilidades encontradas.

Palabras claves: Máquina, Virtual, VM, Virtualización, Ofuscación, Ingenería, Reversa, Seguridad.

CONTENTS

1. INTRODUCTION

1.1 Intellectual property & reverse engineering

A company keeps valuable secrets in its products. In the case of software, a program implements fundamental algorithms, data structures, and protocols. A leak of them can either be an economical loss or a security breach. For instance, a rival company can steal critical logic or malicious users can exploit the product.

The user of a binary program can modify it and control the execution environment. In this way, a user can easily turn into an attacker. We call reverse engineering the process of extracting secrets of a binary.

Considering the risks of vulnerable intellectual property, we define the problem of our work as the protection of intellectual property in software. That is countermeasures against reverse engineering of a binary.

1.2 Countermeasures against reverse engineering

Next we describe a scenario originally written in [4]. It helps us to illustrate possible countermeasures against reverse engineering. Also, we specify which type of countermeasure is the one focused in our work.

Alice is a software developer and she sells her programs to be downloaded. Bob is a rival software developer. He wants to know the algorithms and data structures implemented by Alice. By doing this, he considers he can easily beat Alice in sales.

Bob wants a high level representation of Alice's secrets. Not necessarily the exact source code she wrote, but one that conceptualizes her secrets. Bob starts with the binary code of the application that Alice sells. Also, Bob does not need to reverse-engineer the whole application but just the sections of interest. On the other hand, Alice does not need to protect the whole binary but just the sections she considers confidential.

Simply put, it is a two players game. One player tries to avoid an attack and the other one reverse-engineers the binary.

How can Alice protect her binary? One way is to take legal actions against any reverse engineering attempt. However, this path can be unaffordable for a small developer.

Then, Alice must consider a technical approach. She makes her binary more resilience to an attack. The available options to protect Alice's secrets from Bob are:

- a client-server architecture
- encryption of the binary code
- code obfuscation

1.2.1 A client-server architecture

Alice's business scheme gives the binary with all its secrets to the user. Therefore, the user has physical access to it and ultimately risking all the intellectual property in the program.

If the attacker does not have access to the confidential algorithms and data structures, it becomes impossible to him to know how they work. Following this logic, Alice must change her program architecture to keep all her secrets safe. Instead of giving the original binary, she gives a client program and a server performs all business related computations. The attacker does not have access to Alice's key algorithms.

A possible drawback could be that Alice may not have enough infrastructure to implement this solution. Also, the domain problem can be incompatible with the client-server architecture.

Even if it is feasible, Alice still faces vulnerabilities. The client program is not protected and can still be subject to reverse engineering. Bob can gain knowledge of the protocol between the client and the server. Any vulnerability spotted by Bob can lead to serious issues to Alice. To sum up, we still need to find ways to hamper any reverse engineering attempt performed by Bob.

1.2.2 Encryption of the binary code

Alice can have the actual binary code encrypted. When the application runs, it decrypts the original code and executes it. If the decryption is done by software, Bob can intercept the output of the process. In this way, he has the original and unprotected code.

1.2.3 Obfuscation of the binary code

An obfuscator is a program that has as input a program and as output another one. The input program is transformed in such a way that the output program is functionally equivalent but harder to understand by Bob.

The obfuscation cannot guarantee an unbreakable protection against reverse engineering. Bob still has physical access to the binary code. The intent of the obfuscation is to gain time before an attacker has got knowledge from the program.

In contrast to a client-server architecture, an obfuscation does not prevent Bob having Alice's secrets. It just makes them harder to be understood.

In this work we will address this type of protection in greater depth. We find it more adoptable than modifying the architecture of a program and stronger than encrypting it.

1.3 Related work

Tigress [9] is a free virtualizer for the C language that supports many novel defenses against both static and dynamic reverse engineering. However, it is not open source and it only works for the C language. The devirtualization attack [1] can successfully work on the challenges proposed by Tigress.

There are also other two known virtualizers ([7] and [8]) but neither is open source nor free.

In addition to the VM obfuscation we also develop two others:

1. lookup tables
2. range divider

The first one is based on the fact that current dynamic symbolic execution techniques cannot correctly model input dependant memory accesses. This idea is validated in [11]. We extend that idea by folding instructions into arrays.

The second one is based on the ideas in [2]. The idea is leveraged by finding a concrete way for partitioning the domain of a variable. We also provide benchmarks while [2] provides none. Also, we explain the usefulness of nesting the obfuscation.

1.4 Contributions

Our work involves the development of the next three obfuscations:

1. the virtual machine
2. the lookup tables
3. the range divider

We implement all of them in LLVM. Therefore, they can be used independently of the target architecture and the original programming language.

The first one and most important obfuscation in this thesis is evaluated in a reverse engineering challenge against real reverse engineers. As a result, the obfuscation is resilient against manual attacks. However, it shows to be vulnerable to state-of-the-art attacks such as devirtualization techniques [1].

In order to mitigate its flaws, we design two effective obfuscations (lookup tables and range divider) against [1]. They exploit technical and conceptual problems of dynamic symbolic execution and concolic execution.

1.5 Structure of the thesis

Our work is divided in ten chapters and one appendix.

1. Chapter 1 contains the motivation, contributions, and structure of the current work.

2. Chapter 2 covers the preliminary concepts to understand the thesis. For example, we include a preliminary outline of:
   (a) a virtual machine obfuscation.
   (b) a popular framework that an attacker could use to develop tools against our work. It provides several program analyses.
   (c) a theoretical view of a manual or automatic attack to our protection.
3. Chapter 3 exhaustively describes our VM obfuscation, it is the main protection of the thesis.
4. Chapter 4 explains the designed challenge to stress the VM obfuscation. We also mention the fundamental feedback we got to enhance the obfuscation.
5. Chapter 5 defines a complementary obfuscation called lookup tables and in chapter 6 it is evaluated.
6. Similarly in chapter 7, we define the complementary range divider obfuscation. It is evaluated in chapter 8.
7. Chapter 9 defines the performance benchmark of the VM obfuscation.
8. Chapter 10 mentions possible future work.
9. Lastly, the appendix contains big tables and large source code that was placed there to ease the reading of this document.

2. BACKGROUND

2.1 A virtual machine obfuscation in a nutshell

A virtual machine obfuscation transforms a function **F** in the following way:

1. It generates an interpreter (also known as virtual machine) for a custom set of instructions.
2. It translates **F** into a semantically equivalent sequence of instructions (also known as bitcodes) for the interpreter.
3. Replaces the body of **F** with an invocation to the interpreter to process the newly created bitcodes.

An interpreter is a function that receives as input a sequence of instructions. Then, it executes each instruction and updates its state. It behaves similarly as a CPU implemented in software. The interpreter can be implemented as a loop processing an array of instructions.

2.1.1 An example

We present a function to obfuscate. Then, we show how it is translated to a sequence of instructions for the interpreter. Also, how the interpreter works and lastly the final form of the function to obfuscate.

Function to obfuscate

Suppose we have to obfuscate the function in Listing 2.1.

```
int32_t abs(int32_t x){
    if (x >= 0)
        return x;
    int32_t res = x * -1;
    return res;
}
```

Listing 2.1: Absolute value function

The interpreter

The obfuscator creates an interpreter. The interpreter is a function that processes a sequence of instructions (bitcodes).

As we previously said, it has a resemblance of CPU implemented in software. Here, we show a C/C++ pseudo-code of it.

```
int32_t virtual_machine(Registers regs,
                                        Bitcodes bitcodes){
  ProgramCounter pc;

  while (bitcodes.hasInstructions(pc)){
    Instruction ins = bitcodes.instruction(pc);

    switch(ins.getOpcode()){
      case MUL:
        int32_t mul = regs.load_register(ins.operand(0))*
          regs.load_register(ins.operand(1));
        regs.store_register(ins.operand(2), mul);
        pc.Increase(ins.size());
        break;
      case RET:
        return regs.load_register(ins.operand(0));
      case JUMP_COND:
        // updates program counter accordingly
        break;
      case GREATER_EQUAL:
        bool r = regs.load_register(ins.operand(0)) >=
        regs.load_register(ins.operand(1));
        regs.store_register(ins.operand(2), r);
        pc.Increase(ins.size());
        break;
    }
  }
}
```

Listing 2.2: A pseudo-code for the interpreter (aka virtual machine).

The interpreter maintains a program counter that dictates which instruction is the next to process. It also has a set of registers that hold preloaded values as well as intermediate computations.

It reads the opcode of the next instruction and execute the appropriate routine associated with it. Then, updates the program counter and repeats the process. The interpreter will be executed instead of the original code of the obfuscated function.

The sequence of bitcodes for the interpreter

The body of the function to be obfuscated is translated into a sequence of bytes. Those bytes are the new instructions for the interpreter. They must follow certain encoding, so the interpreter can process them correctly. The interpreter is tied to a custom set of instructions.

In the context of this example, we can suppose the buffer containing the sequence of instructions is: 0x04030403000100030605020202010000. In Table 2.1, we show how we interpret each byte. The table must be read per row beginning by the one at the top. Each row must be read from left to right. The first cell of the table corresponds to the last hex digit of the buffer's constant. The second cell corresponds to the penult hex digit and so on.

opcode	register id	register id	register id	size (bytes)
0x00 greater or equal	0x00 operand	0x01 operand	0x02 destination	4
0x02 conditional jump	0x02 boolean condition	0x05 offset_true	0x06 offset_false	4
0x03 return	0x00			2
0x01 mul	0x00 operand	0x03 operand	0x04 destination	4
0x03 return	0x04			2

Tab. 2.1: Byte interpretation of the buffer containing the bitcodes

Result of the obfuscation

Here, we show a pseudo-code that represents the result of the obfuscation. The original function's body is removed and replaced with a invocation to the interpreter. This is the code an attacker would see.

```
int32_t abs(int32_t x){
        Bitcodes bitcodes(0x040304030001
                                000306050202020100000);
        Registers registers(7);
        // set initial values in registers
        // ie: constants
        return virtual_machine(registers, bitcodes);
}
```

Listing 2.3: A pseudo-code for the obfuscated function.

2.2 LLVM

Fig. 2.1: LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Among many features, LLVM provides a framework for program analysis and transformation. The code to be analyzed or transformed is represented in an intermediate language, called *LLVM-IR*. That representation is used to communicate between LLVM components. Those components are *frontends* and *backends*. The first ones transform high level source code into *LLVM-IR* instructions while the second ones transform the *LLVM-IR* into machine code. Like this, different high level languages – as C or Fortran – can be translated to the *LLVM-IR* and be optimized using the same infrastructure.

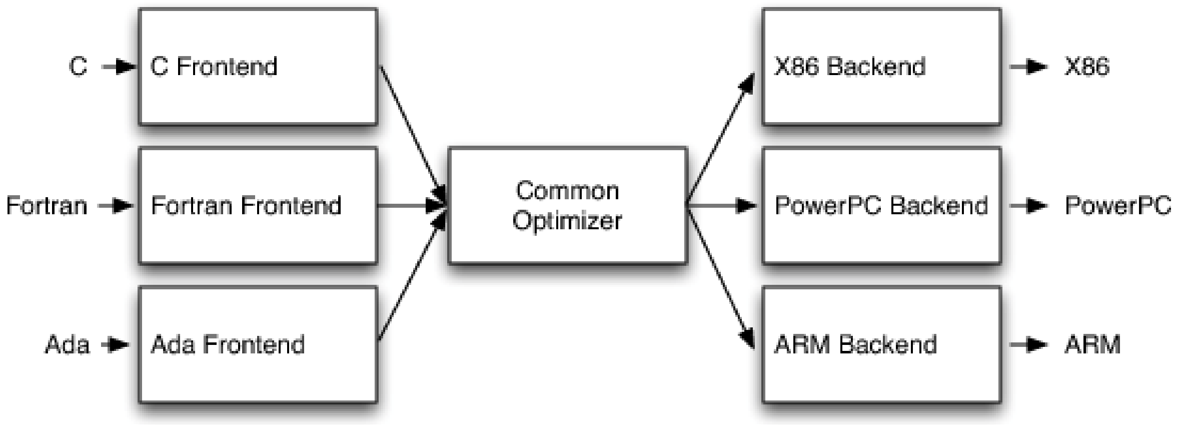


Fig. 2.2: LLVM architecture

In Figure 2.2, there are several *frontends*. Each of them outputs a program in *LLVM-IR*. Then, the implemented optimization is re-used independently of the original high level language. The compiler optimization outputs a program in *LLVM-IR*. The different *backends* generate the binary for each specific platform.

2.2.1 LLVM pass

A *LLVM pass* is a transformation or analysis of *LLVM-IR* code. It has as input and as output *LLVM-IR* code, in this way they can be pipelined. In Figure 2.3, we have an example of how different *LLVM passes* pipeline all together. The *LLVM optimizer* is responsible of executing *LLVM passes*. They are also knwon as optimizations. Our obfuscations are implemented as *LLVM passes*.

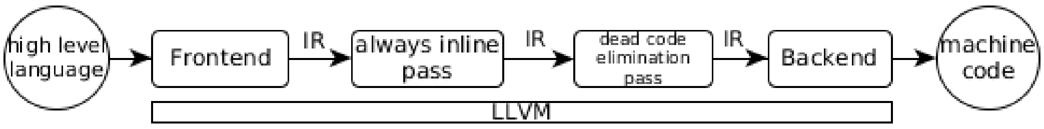


Fig. 2.3: Pipeline of LLVM passes

2.3 Triton: a dynamic binary analysis framework

TRITON
Dynamic Binary Analysis

Triton is a dynamic binary analysis framework. It offers features such as:

- a Dynamic Symbolic Execution engine (DSE).
- a Dynamic Taint Analysis engine.
- a tree semantic representation of the x86 and x86-64 instruction set.
- An interface to interact with a SMT-solver.
- Python bindings for every feature of the framework.

Triton allows its users to develop reverse engineering tools in top of it. In the following sections, we describe the analyses Triton provides. The explanations shown here are heavily based on the official wiki of the project.

2.3.1 Static and Dynamic Symbolic Execution

Before explaining Dynamic Symbolic Execution, we briefly introduce the concept of Static Symbolic Execution.

Static Symbolic Execution analyzes a program without executing it, it is an offline procedure. It processes each program statement and only keeps a symbolic state of the program. The symbolic state at a certain program point expresses all possible concrete values of variables up to that point. Those concrete values are the ones generated by every possible path to the chosen point.

This technique can potentially generate exponential formulas in size because it abstracts every possible path of a program. It is also unfeasible to automatically determine the abstract state of a program after a loop. Working with such big and complex formulas can exhaust current SMT solvers. Therefore, its usefulness is limited when it comes to checking properties on those formulas. Usually, the formulas generated by static symbolic execution are an over approximation of the original program behavior. In this way, they introduce unexisting traces in order to remain sound.

Consider the following fragment of a program:

```
if (x > 2)
    y = x * 2;
else
    y = x + 5;
```

Fig. 2.4: Conditional program

The program symbolic state right after the if-else conditional is: $(x > 2 \Rightarrow y = 2*x) \wedge (x \leq 2 \Rightarrow y = x + 5) \wedge (x = \varphi)$. Where φ is a formula representing the value of x up to that point.

The Dynamic Symbolic Execution relaxes the objective of deducing a symbolic state capturing every possible concrete state up to certain point. In order to do so, it performs an online analysis. It executes the program with a particular input. While the program executes, exercising a single program path, it generates symbolic states for each executed program point. Those symbolic states are only representative of all actual traces sharing the same path constraints (if evaluations/conditional jumps). The advantage is that the generated formulas (the symbolic states) are simpler and easier to deal with for SMT solvers.

Consider again the previous example, but now assuming that the chosen input makes x evaluate to 1. Then, the symbolic state after the if-conditional would only consider that path constraint. The formula is:$(y = 5 + x) \wedge (x = 1)$.

Another use is as follow. Suppose you ran your program with a particular input while performing DSE and gathering all taken path constraints. Then, you can ask an SMT solver for a new input not satisfying them. That is possible because every time a compare instruction is executed a formula is available for the operands that are being used. Those formulas are expressed in terms of the input of the program. We can perform again DSE with the new input. Repeating the process we can recover multiple paths of the program. If there are no loops, we could retrieve all program paths. There are several strategies to perform this procedure and we call it concolic execution.

2.3.2 Dynamic Symbolic Execution (DSE) in Triton

The Dynamic Symbolic Execution engine allows to maintain symbolic states for the registers and the memory at each execution point. The engine keeps:

- a map for the symbolic states of the registers

- a map for the symbolic states of the memory
- a set containing symbolic references

During the execution, the table for the registers's symbolic states is updated according to the semantic of each instruction.

The table models the mapping $\langle rid \rightarrow \varphi_x \rangle$ for every register, where rid uniquely identifies a register and φ_x identifies only one symbolic expression (formula in integers logic).

Simply put, at each point of the program, every register points to its symbolic expression that represents the semantic of its last assignment. In Table 2.2, there is an example of how registers point to symbolic expressions.

Step	Register	Instruction	Set of symbolic expressions
init	$eax \rightarrow nothing$	none	$\emptyset$
1	$eax \rightarrow \varphi_1$	mov eax, 0	$\{\varphi_1 = 0\}$
2	$eax \rightarrow \varphi_2$	inc eax	$\{\varphi_1 = 0, \varphi_2 = \varphi_1 + 1\}$
3	$eax \rightarrow \varphi_3$	add eax, 5	$\{\varphi_1 = 0, \varphi_2 = \varphi_1 + 1, \varphi_3 = \varphi_2 + 5\}$

Tab. 2.2: Symbolic tracking of eax during execution.

Similarly, the symbolic states of the memory are tracked. It is modeled by the mapping $\langle addr \rightarrow \varphi_x \rangle$. Where $addr$ represents a memory address and φ_x represents the only reference of a symbolic expression. In Table 2.3, we have an example for memory addresses.

Step	Register	Memory	Instruction	Set of symbolic expressions
1	$eax \rightarrow \varphi_1$	no changes	mov eax, 0	$\{\varphi_1 = 0\}$
2	no changes	$sp \rightarrow \varphi_2$	push eax	$\{\varphi_1 = 0, \varphi_2 = \varphi_1\}$
...	...	...	...	...
10	$ebx \rightarrow \varphi_3$	no changes	pop ebx	$\{\varphi_1 = 0, \varphi_2 = \varphi_1, \varphi_3 = \varphi_2\}$

Tab. 2.3: Symbolic tracking of registers and memory.

From the previous example, it is possible to deduce that $ebx = eax = 0$.

What would happen if a register or memory address does not have associated a symbolic expression when it is used as an operand? Triton creates a symbolic expression of a constant value. That value is the one that the register or memory address holds in the actual execution. We call this concretization of a value. In this way, the analysis is capable of starting at any point of the program.

This is relatable to loading to a register the value in an array of constants. There was no instruction computing those values, they came from the binary file. Thus, Triton is not able to have a symbolic expression associated to that memory address. In other words, Triton does not know how they were built.

In chapter 5, we describe how to use this behavior against Triton.

2.3.3 Symbolic expressions represented as trees

Triton stores symbolic expressions as trees. The similarity between trees and symbolic expressions is that both are recursive. That's that one tree is composed by other sub-trees and a symbolic expression can also be composed by multiple sub-expressions.

The possible values in a register are not mathematical integers. In terms of a SMT solver, they are bit-vectors of certain size. The same logic follows for mathematical operations between registers. The addition between two register cannot be directly mapped to the addition between two integers. Therefore, when we construct the symbolic expression of a register we must refer to bit-vector operations. The symbolic expression resembles the logical terms that a SMT solver uses to build a formula.

Consider the following sequence of instructions processed using DSE:

1	mov al, 1
2	mov cl, 10
3	mov dl, 20
4	xor cl, dl
5	add al, cl

Tab. 2.4: Assembly instructions

After executing instruction #5, the tree of register AL is similar to this:

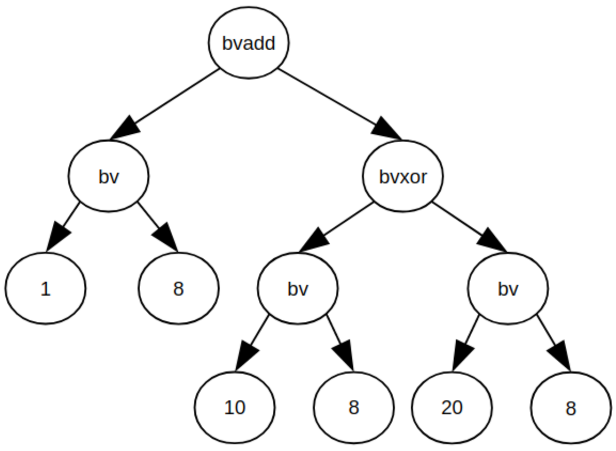


Fig. 2.5: Tree of register AL after execution of instruction #5

At the leaves of the tree we have *bv* (bit-vector) nodes. They represent the constant values of 1, 10 and 20 in bit vectors of 1 byte. A bit vector is represented as a binary tree with its left child node indicating the value stored and the right child node representing the size in bits.

Then, the expression constructed by the xor instruction, is represented by the subtree from *bvxor*.

At the root of the tree, we have the add operation represented by node *bvadd*.

2.3.4 Manipulating a symbolic expression

Consider again example in Table 2.4 and assume we have processed the entire sequence of instructions.

Suppose you want to find out which possible value in register AL at instruction #1 could have made the final addition (instruction #5) return 80.

The symbolic expression pictured in Figure 2.5 is a formula in integers logic. It has the particularity that there is no variable, it is a formula made of constants. Conceptually speaking, it represents: $1 + (10 \oplus 20)$.

In order to solve our question about register AL, we need to express the original value of AL as a variable in the previous formula and equal that expression to 80. The formula we want to solve now is: $X + (10 \oplus 20) = 80$.

Triton offers an interface to manipulate symbolic expressions in such a way you can build your own equations. Then, you can send the formula to a SMT solver and it will try to find suitable instantiations for the introduced variables (in our case X).

The formula to be sent to the SMT solver is pictured in Figure 2.6. The original tree for the result of instruction #5 is changed. It has an X (a symbolic variable) instead of a constant 1 as the original value of register AL. Also, the tree is appended as a left child of an equal tree. The right child of the equal tree is the constant 80 expressed in 1 byte.

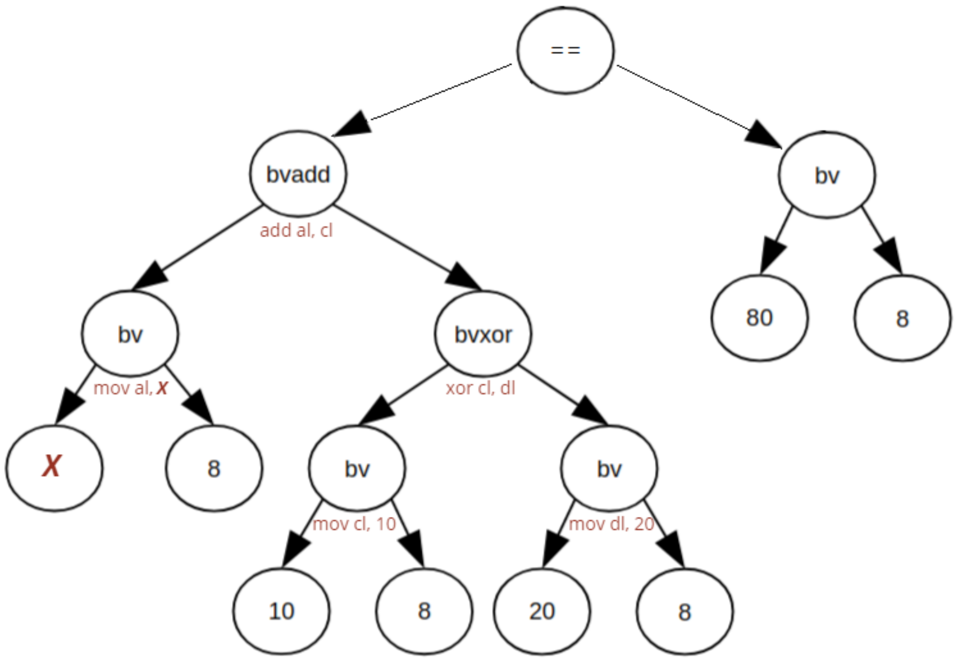


Fig. 2.6: Query to SMT solver.

2.3.5 Taint engine

The purpose of the dynamic taint analysis is to keep track of the information flow. The analysis starts from a source (usually user inputs) up to certain points (ie. conditional instructions). Conceptually, the source taints the registers, memory and instructions through which it flows during execution.

Therefore, this analysis enables the understanding of which region of the memory and registers are controllable by the user's input. We can test if an application's control flow can be compromise by the user's input, very useful for security reasons.

The taint analysis has a taint policy which is made of three properties: how a new taint is introduced, how the taint propagates and how the taint is verified during the execution.

Taint introduction. It specifies how the taint is introduced. Generally, one considers all variables (registers, memory and instructions) untainted at the beginning of the analysis. Then, the taint is introduced when the user's input is processed. For instance, you can flag that certain method always returns a tainted value (ie: scanf in C).

Taint propagation rules. It determines when data is tainted. The current data can be dependant of already tainted data. Suppose that d_3 is calculated from data d_1 and d_2. Usually, d_3 is considered tainted if at least one of d_1 or d_2 is tainted too.

Taint checking rules. This rule is dependant of the analysis's user. For instance, a module that checks for an attack can stop the execution of the program in case the operand of one jump is tainted.

Triton's taint engine implements propagation rules independently of any architecture. Also, it offers an API for the user to define introduction and checking rules.

Taint propagation rules trade-offs

While implementing the rules, a decision must be taken between precision and performance. There are three possible choices:

- Over-approximation
- Precise-approximation
- Under-approximation

Triton implements an over-approximation that has the following advantages over precise-approximation:

- Easier to implement.
- Small runtime overhead.
- Low memory consumption.

An over-approximation is the one that best fits in a bit-level granularity. Suppose the following scenario:

Let r be a register of 16 bits with the configuration [x-x-x—x-xx-x-x]. The x are the bits controllable by the user and the - are the bits that are not. Assume that this state of the register is the result of an arithmetic computation. It can change depending of the concrete values used as user's input.

The user-controllable bit information cannot be generalized for any other concrete input. That information is computed based on the very specific concrete value of the current input.

Although, you can conclude that for any other concrete input that makes the program take the same path, you can control that register or memory location.

Imagine that you want to know how to change the input in such a way that you obtain a needed concrete value in a controllable register at a certain point. For instance, you may want to force the program to take a specific branch.

Even if you spend resources computing a perfect taint analysis, you won't be closer to answer your problem. If you compute a sub-approximated analysis you may miss a controllable register.

The valuable information is that taint analysis can tell you if a register or memory location is controllable and an over-approximation is good enough. You still need to find a way to find out a formula that expresses the values in that register. If that formula has the user input as a variable, you can use an SMT solver to find an answer.

Precision with performance costs

An over-approximation can sacrifice precision in exchange of simplicity and performance. We will follow the next ASM code.

1	mov ax, 0x1122	RAX is untainted
2	mov al, byte ptr [user_input]	RAX is tainted
3	cmp ah, 0x99	¿can the user control this statement?

In a over-approximation, the taint engine generates a false positive (AH is tainted when it should not) for this example. The 7 bits of RAX (RAX[7..0]) are tainted and the rest are not (RAX[63..8]).

Consider a case where an attacker develops an exploit for an executable. The attacker intention is to know if a register at a specific execution point can be controllable by him and what values he can make the register hold.

A taint analysis does not gives a full answer. In order to fix this issue, one can use the power of the symbolic execution and ask a model (to a SMT solver). This is a much costly operation. However, the user has the option to decide when to pay the performance penalty at the sake of precision.

Triton uses symbolic execution to provide precision and over-approximated taint analysis to know when ask a model to a SMT solver. When the user asks for the model, we can precisely know which values can hold a tainted register in terms of the symbolic variables.

2.4 Devirtualization of a binary

A devirtualization technique reverts the transformation done by a virtual machine obfuscation (introduced in Section 2.1).

In a virtualized function we have a buffer containing the virtual instructions for the interpreter. In the binary, the interpreter is in machine language and the buffer contains plain bytes. Those bytes follow an encoding expected by the interpreter. When the interpreter processes that buffer produces a result equivalent to the one when it was not virtualized.

In a devirtualization technique, we want to translate the buffer's bytes into assembly code again. In that way, we don't have to rely on an interpreter that processes those bytes. We can directly dive into the assembly to understand the intent of the obfuscated function. In the assembly there is no interpreter anymore, we removed a layer of protection.

A devirtualization can be either automatic or manual. Here, we describe both. We summarize the explanation found in [1].

2.4.1 Manual devirtualization

Typically, this approach implies gathering full knowledge of the virtual architecture and writing a dissasembler targeting it. The dissasembler takes as input the interpreter's buffer (containing virtual instructions) and outputs assembly code equivalent to the one in the buffer (there is no interpreter anymore). The reverser should invest time in reverse engineering the virtual machine. The difficulty of this task is dependant of the reverser expertise.

In order to write a dissasembler for the virtual architecture, the reverse must be able to:

- Identify which parts of a program are virtualized.
- Identify each component of the VM (virtual handlers, virtual program counter, etc).
- Establish correspondance between handlers and their opcodes (to understand their semantics). Also, understand where their operands are stored (virtual registers).
- Understand how the virtual program counter is increased.

Based on all this knowledge one can know the semantics of each instruction in the buffer and how they are parsed by the VM. Thus, it is possible to write a dissasembler for this particular architecture.

However, this task can be time consuming and very subject to the reverser expertise. How to solve each previous step is not trivial and there is no general way to address it.

2.4.2 Automatic devirtualization

The following is a summarization of the work in [1].

Intuition

Consider the case in which you have a binary and its virtualized version. You run them both for a particular input and keep track of the assembly instructions executed. Let T be a trace of the non virtualized binary and T_{vm} the virtualized trace version.

What would be the difference for the same input between T and T_{vm}? In T_{vm} you have assembly instructions exercising the interpreter's machinery (we call virtual machine and interpreter indistinctly), for instance increasing the virtual program counter. In T there is no counterpart for them, because that was not part of the original behavior. Although, T and T_{vm} have counterparts for assembly instructions that are part of the result. Imagine the binary is computing factorial, in both traces you must have assembly instructions doing a multiplication somehow. Also, the multiplication is used to compute the result in both cases.

Intuitively, if in T_{vm} we can tell which assembly instructions are only there as the virtual machine's machinery, we could filter them out and obtain a new trace in which the virtual machine obfuscation is removed. This only holds to reconstruct one path of the original program. If we cover all possible paths, we are able to reconstruct the whole program without any protection. This is feasible for programs in which there are a finite number of paths (ie: hashes).

Example

Consider the code fragment in Listing 2.4. SECRET is a function that is virtualized in our executable. Integer $k1$ and $k2$ have intermediate computations that are used to calculate the returned value.

Suppose we get a trace of SECRET under some input i. Let Table 2.5 be such trace. In Table 2.5 we should have assembly instructions, however we will express each item conceptually for clarity reasons. That's one item in the table can be composed by many assembly instructions. Also, assume the trace is related to the processing of the $k1$ addition (one virtual machine execution cycle).

Instructions in items 1,2,3,4, and 6 are related to the virtual machine implementation. Neither of them affect or are part of the returned value by SECRET. Oppositely item 5 is part of the result and it is dictated by the original code.

The VM's handler corresponding to the ADD instruction not necessarily performs an add instructions (at assembly level), because it can be also obfuscated. However, the obfuscation does not alter the fact that instructions of item 5 compose the returned value of SECRET. In addition, instructions of item 5 have as operands values dependant of the SECRET's input.

```
int SECRET(int input){
        // ...
        int k1 = prev_comp_1 + prev_comp_2;
        int k2 = prev_comp_3 + prev_comp_4;
        // ...
}
```

Listing 2.4: Partial code of original program P

Tab. 2.5

item number	conceptual trace item	source
1	reading_opcode_inst	VM's machinery
2	jump_to_handler_inst	VM's machinery
3	reading_operand_inst	VM's machinery
4	reading_operand_inst	VM's machinery
5	add_inst	original code semantic
6	update_vpc_inst	VM's machinery

Overview

The main steps in the devirtualization method are (illustrated in Figure 2.7):

- **Step 0**: Identify input.
- **Step 1**: On a trace, isolate pertinent instructions using a dynamic taint analysis.
- **Step 2**: Build a symbolic representation of these tainted instructions.
- **Step 3**: Perform a path coverage analysis to reach new tainted paths.
- **Step 4**: Reconstruct a program from the resulting traces and compile it to obtain a devirtualized version of the original code.

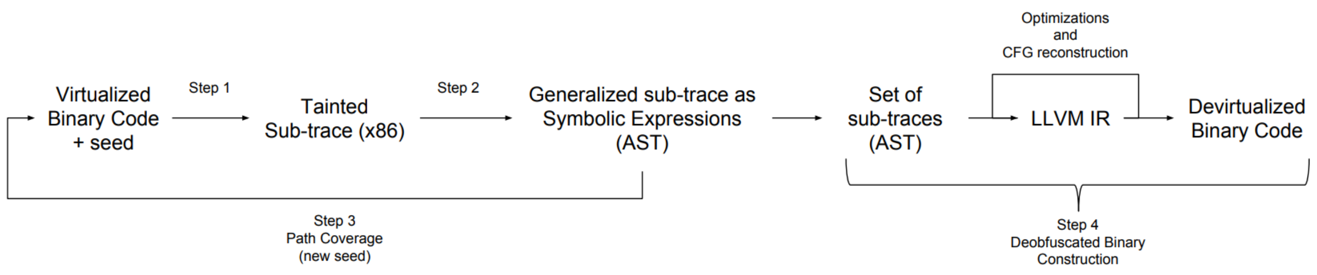

Optimizations and CFG reconstruction
Virtualized Binary Code + seed
Step 1
Tainted Sub-trace (x86)
Step 2
Generalized sub-trace as Symbolic Expressions (AST)
Set of sub-traces (AST)
LLVM IR
Devirtualized Binary Code
Step 3 Path Coverage (new seed)
Step 4 Deobfuscated Binary Construction

Fig. 2.7: Schematized Approach

This process heavily relies on Triton (previously introduced) and its analyses. Next is a description of each step:

Step 1 - Dynamic Taint Analysis

The first step aims at separating those instructions which are part of the virtual machine internal process from those which are part of the original program behavior. In order to do that, we taint every input of the virtualized function.

Running a first execution with an input generated by a random seed, we get as a result a subtrace of tainted instructions. We call these instructions: pertinent instructions. They represent all interactions with the inputs of the program, as non-tainted instructions have always the same effect on the original program behavior. At this step, the original program behaviors are represented by the subtrace of pertinent instructions.

But this subtrace cannot be directly executed, because some values are missing, typically the initial values of registers.

Step 2 - A Symbolic Representation

The second step abstracts the pertinent instruction subtrace in terms of a symbolic expression for two goals: (1) prepare DSE exploration, (2) recompile the expression to obtain an executable trace. In symbolic expressions, all tainted values are symbolized while all un-tainted values are concretized. In other words, our symbolic expressions do not contain any operation related to the virtual machine processing (the machinery itself does not depend on the user) but only operations related to the original program.

Step 3 - Path Coverage

At this step we are able to devirtualize one path. To reconstruct the whole program behavior, we successively devirtualize reachable tainted paths. To do so, we perform path coverage [6] on tainted branches with DSE. At the end, we get as a result a path tree which represents the different paths of the original program (Figure 2.8). Path tree is obtained by introducing if-then-else construction from two traces t_1 and t_2 with a same prefix followed by a condition C in t_1 and $\neg C$ in t_2.

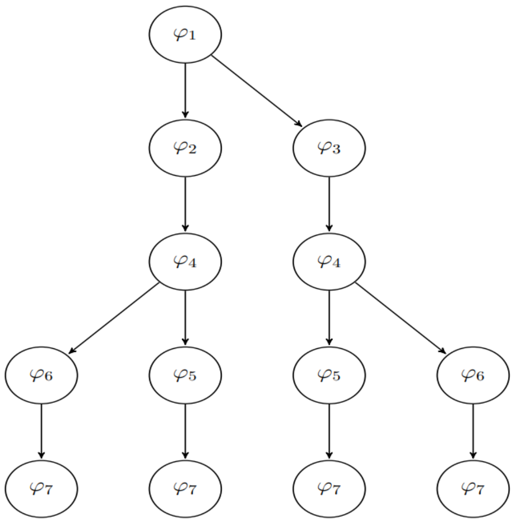


Fig. 2.8: Path tree

Step 4 - Generate a New Binary Version

At this step we have all the information to reconstruct a new binary code: (1) a symbolic representation of each path; (2) a path tree combining all reachable paths. In order to produce a binary code we transform our symbolic path tree into the LLVM IR to obtain a LLVM Abstract Tree (Fig. 2.7) and compile it. In particular we benefit from all LLVM (code level) optimizations to partially rebuild a simplified Control Flow Graph (Figure 2.9). Note that moving on LLVM allows us to compile the devirtualized program to another architecture. For instance, it is possible to devirtualize a x86 function and compile it to an ARM architecture.

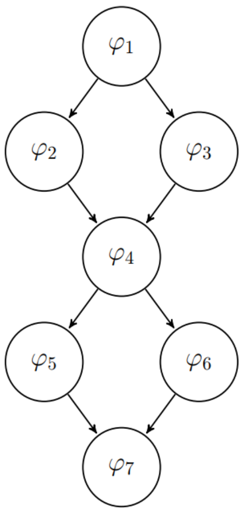


Fig. 2.9: A reconstructed CFG

2.5 IDA Disassembler

IDA Disassembler is a professional tool used for reverse engineering and debugging of binaries. Here, we briefly introduce its main functionality that is disassembling a binary. Simply put, it translates machine code into assembly language.

We show how it works for a simple binary. It is the absolute value function shown in Listing 2.1.

We are going to show how IDA works for the following tasks:

- Disassembling the binary
- Reconstructing the control flow graph of the binary.

Also, we show the same results for the obfuscated version of Listing 2.1.

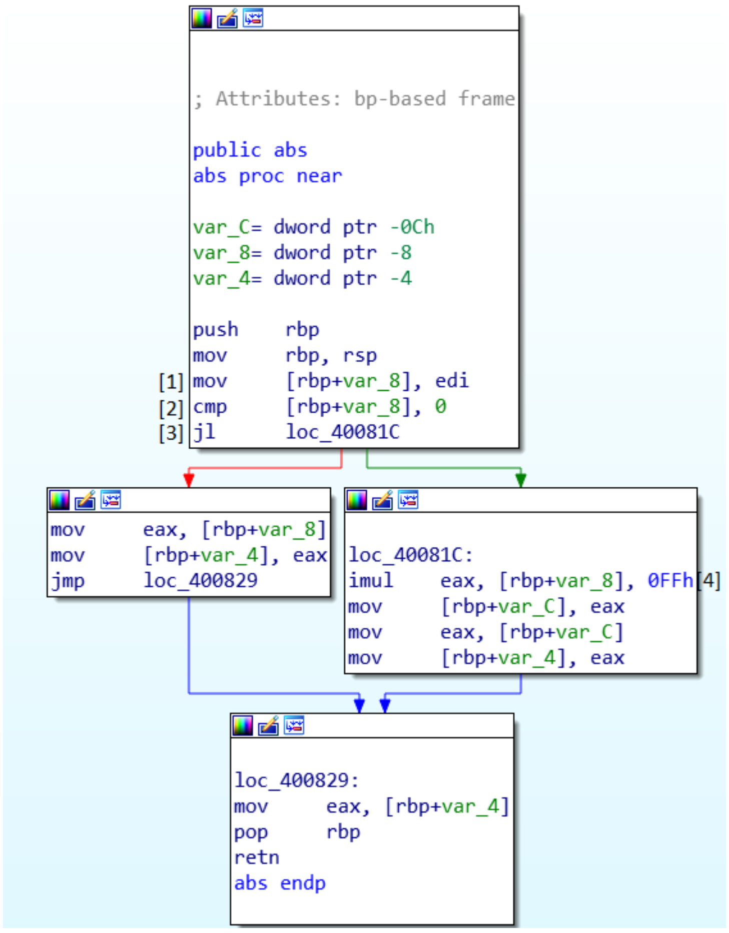
2.5.1 An example: a non-obfuscated binary

We compile the code in Listing 2.1 with no optimization. The reason is that we just want to focus on the features of IDA. The reconstructed control flow graph is in Figure 2.10. The image has 3 marks that are mentioned next.

In mark **(1)**, we move the input to a variable in the stack. Then, in **(2)** we compare that value against 0. In **(3)**, we perform a jump if the input value is lower than 0.

The red arrow indicates the jump target if the condition evaluates to false. In that basic block, we just move the value to one register and then again to another stack variable. Lastly, in the final basic block we move the input value to the eax register. This path of the program is redundant because we compiled it without optimizations.

The green arrow indicates the jump target when the input is lower than zero. In mark [4] we perform the multiplication to -1. Then, after a series of move instructions we place the result in eax.



```
; Attributes: bp-based frame

public abs
abs proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

    push    rbp
    mov     rbp, rsp
[1] mov     [rbp+var_8], edi
[2] cmp     [rbp+var_8], 0
[3] jl      loc_40081C
```

```
mov     eax, [rbp+var_8]
mov     [rbp+var_4], eax
jmp     loc_400829
```

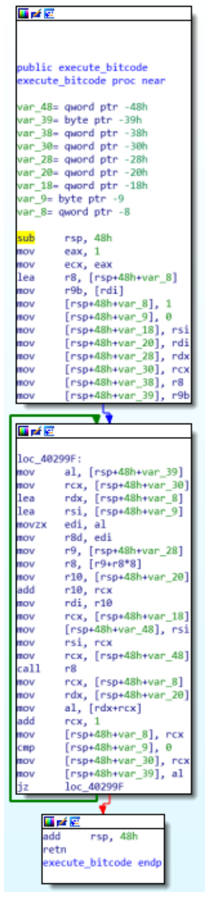
```
loc_40081C:
imul    eax, [rbp+var_8], 0FFh [4]
mov     [rbp+var_C], eax
mov     eax, [rbp+var_C]
mov     [rbp+var_4], eax
```

```
loc_400829:
mov     eax, [rbp+var_4]
pop     rbp
retn
abs endp
```

Fig. 2.10: Assembly code and control flow reconstructed by IDA

2.5.2 An example: an obfuscated binary

Here, we show the control flow and assembly code for the same original source code but obfuscated with the virtual machine.



```
public execute_bitcode
execute_bitcode proc near

var_48= qword ptr -48h
var_39= byte ptr -39h
var_38= qword ptr -38h
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_9= byte ptr -9
var_8= qword ptr -8

sub     rsp, 48h
mov     eax, 1
mov     ecx, eax
lea     r8, [rsp+48h+var_8]
mov     r9b, [rdi]
mov     [rsp+48h+var_8], 1
mov     [rsp+48h+var_9], 0
mov     [rsp+48h+var_18], rsi
mov     [rsp+48h+var_20], rdi
mov     [rsp+48h+var_28], rdx
mov     [rsp+48h+var_30], rcx
mov     [rsp+48h+var_38], r8
mov     [rsp+48h+var_39], r9b
```

```
loc_40299F:
mov     al, [rsp+48h+var_39]
mov     rcx, [rsp+48h+var_30]
lea     rdx, [rsp+48h+var_8]
lea     rsi, [rsp+48h+var_9]
movzx   edi, al
mov     r8d, edi
mov     r9, [rsp+48h+var_28]
mov     r8, [r9+r8*8]
mov     r10, [rsp+48h+var_20]
add     r10, rcx
mov     rdi, r10
mov     rcx, [rsp+48h+var_18]
mov     [rsp+48h+var_48], rsi
mov     rsi, rcx
mov     rcx, [rsp+48h+var_48]
call    r8
mov     rcx, [rsp+48h+var_8]
mov     rdx, [rsp+48h+var_20]
mov     al, [rdx+rcx]
add     rcx, 1
mov     [rsp+48h+var_8], rcx
cmp     [rsp+48h+var_9], 0
mov     [rsp+48h+var_30], rcx
mov     [rsp+48h+var_39], al
jz      loc_40299F
```

```
add     rsp, 48h
retn
execute_bitcode endp
```

Fig. 2.11: Assembly code and control flow reconstructed by IDA

In Figure 2.11 we have the disassembling of an obfuscated binary. It is the same absolute value function of the previous section, but this time is protected with the virtual machine obfuscation.

The original code is gone and instead we only recover the execution loop of the interpreter (aka virtual machine). The loop is shown by the green arrow. It processes each instruction and then indirectly jumps to the correct handler for the instruction's type. We call handler to the routine that updates the virtual machine's state accordingly to the instruction's semantic.

3. THE VIRTUAL MACHINE OBFUSCATION

3.1 What is a VM?

In the context of our obfuscation, the VM can be thought as an interpreter of a custom set of instructions. Those instructions are called bitcodes. In addition, the interpreter has a set of registers and also access to a memory address space.

Optionally, the VM can also be thought as a CPU implemented in software. From this perspective we can derive it has a virtual architecture and organization.

When the VM starts it reads the first bitcode, then it executes the routine associated with the type of the instruction. The routine has the logic to update the VM's state accordingly to the instruction's semantic. Lastly, the program counter increases (it points to the buffer containing the bitcodes). The process repeats until a return bitcode is reached. The instruction processing loop is called the execution cycle of the VM.

3.1.1 Example

Consider the VM starts the execution on the following bitcode sequence (Table 3.1):

bitcode	semantic
add r2, r1, r0	destination register is r2 and operands are r1 and r0
store r3, r2	memory address in r3 is updated by r2's value

Tab. 3.1: Example of a bitcodes sequence

Based on the PC (program counter), the VM reads the add bitcode. The instruction indicates that r2's value becomes the addition between the value in r1 and r0. The handler is in charge of updating the VM state as intended. The process is repeated until the halt bitcode is reached.

After processing the first bitcode the state of the VM is illustrated in Fig. 3.1

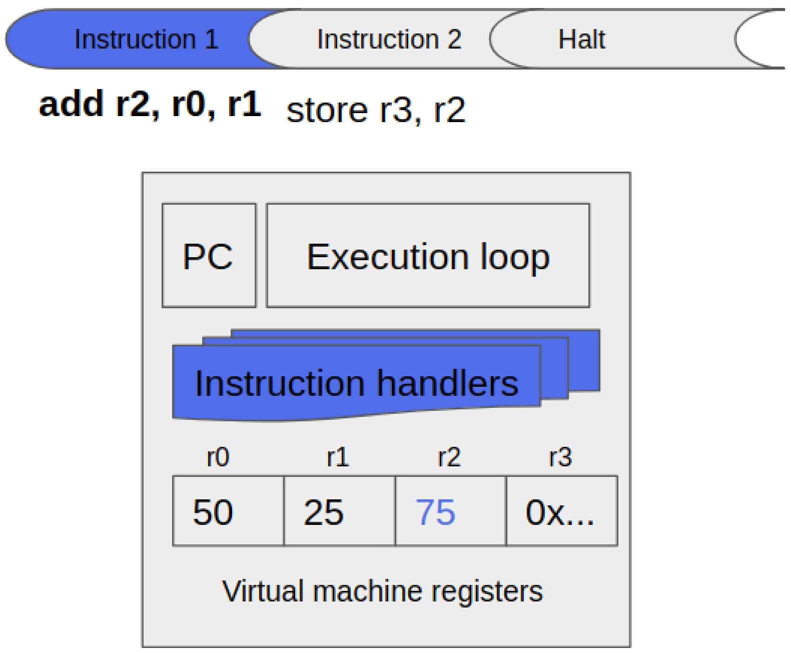


Fig. 3.1: The VM after processing the first bitcode

The executed steps are highlighted in blue:

1. Read the next bitcode

2. Execute the handler for the bitcode

3. Update the VM's state

3.2 Architecture of the VM

3.2.1 Instruction layout

The virtual machine processes instructions (bitcodes) from a buffer. Those instructions are encoded following a specific layout. In other words, its the format the virtual machine expects them to be.

Every instruction has a fixed size determined by the type of operation it is performing. The first field you read from an instruction is the type of it, that is its opcode. Always, there is one byte allocated for it. Then, you have one field per each register used as operand. Those fields hold register ids and they have two bytes assigned. The number of operands per instruction type is fixed. There is no instruction type with a variable number of operands.

type field	size
operation id	1 byte
register id	2 bytes

Tab. 3.2: Size of fields

3.2.2 Registers

Each register is a 64 bits value uniquely identified. Since a register id is 2 bytes there can be at most 65536 registers allocated.

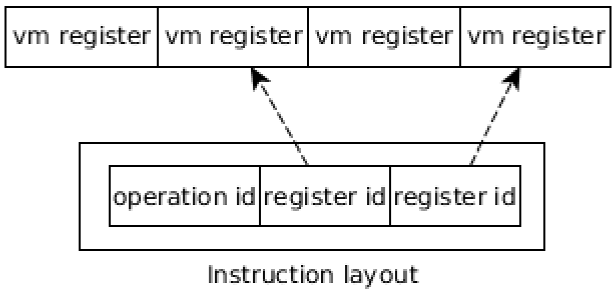


Fig. 3.2: Layout of an instruction

3.2.3 Specification of bitcodes

In this subsection, we describe the intended effect of each operation.

An essential aspect of the VM is that only supports bitcodes for integer numbers. Floating point number operations are not available. We focus our work on hash algorithms, where usually only integers are used.

In order to express the semantic of each operation, we require notations for certain concepts. We define them in Tables 3.3 and 3.4. Finally, we define each bitcode's semantic in Table 3.5.

notation	assignment's left side
r_i	the entire register i is the destination (all 64 bits)
$r_i[r_k]$	least significant r_k bits of register i are the destination
$*r_i[r_k]$	a store of r_k bytes from the memory address indicated by the value of register i

Tab. 3.3: Semantics for expressions in the left side of an assignment

notation	assignment's right side
r_i	value in register i
$r_i[r_k]$	least significant r_k bits from the value in register i
$*r_i[r_k]$	a load of r_k bytes from the memory address indicated by the value of register i
$u\_clean(r_i, r_j)$	extracts the least significant $64 - r_j$ bits from r_i and zero extends them to 64 bits
$s\_clean(r_i, r_j)$	extracts the least significant $64 - r_j$ bits from r_i and sign extends them to 64 bits

Tab. 3.4: Semantics for expressions in assignments

bitcode	semantic
add r_i r_j r_k	$r_i := r_j + r_k$
sub r_i r_j r_k	$r_i := r_j - r_k$
udiv r_i r_j r_k r_q	$r_i := u\_clean(r_j, r_q) \div_{unsigned} u\_clean(r_k, r_q)$
sdiv r_i r_j r_k r_q	$r_i := s\_clean(r_j, r_q) \div_{signed} s\_clean(r_k, r_q)$
urem r_i r_j r_k r_q	$r_i := u\_clean(r_j, r_q) \%_{unsigned} u\_clean(r_k, r_q)$
srem r_i r_j r_k r_q	$r_i := s\_clean(r_j, r_q) \%_{signed} s\_clean(r_k, r_q)$
shl r_i r_j r_k r_q	$r_i := r_j \ll u\_clean(r_k, r_q)$
lshr r_i r_j r_k r_q	$r_i := u\_clean(r_j, r_q) \gg_{logical} u\_clean(r_k, r_q)$
ashr r_i r_j r_k r_q	$r_i := s\_clean(r_j, r_q) \gg_{arithmetic} u\_clean(r_k, r_q)$
and r_i r_j r_k	$r_i := r_j \wedge r_k$
or r_i r_j r_k	$r_i := r_j \parallel r_k$
xor r_i r_j r_k	$r_i := r_j \oplus r_k$
conditional branch r_i r_j r_k	$pc := r_i\ ?\ r_j : r_k$
unconditonal branch r_i	$pc := r_i$
load r_i r_j r_k	$r_i[r_k] := *r_j[r_k]$
store r_i r_j r_k	$*r_i[r_k] := r_j[r_k]$
equals r_i r_j r_k	$r_i := r_j = r_k$
not equals r_i r_j r_k	$r_i := r_j \neq r_k$
$>_{unsigned}$ r_i r_j r_k r_q	$r_i := u\_clean(r_j, r_q) >_{unsigned} u\_clean(r_k, r_q)$
...	...
$\geq_{signed}$ r_i r_j r_k r_q	$r_i := s\_clean(r_j, r_q) \geq_{signed} s\_clean(r_k, r_q)$
gep r_i r_j r_k r_q	$r_q := r_j x r_k + r_i$
ret	
zext r_i r_j r_k	$r_j := u\_clean(r_i, r_k)$
sext r_i r_j r_k	$r_j := s\_clean(r_i, r_k)$
move r_i r_j	$r_i := r_j$
call r_i r_j ... r_k	invoke r_i with arguments r_j ... r_k

Tab. 3.5: Semantic of each supported instruction

3.3 Organization of the VM

In this section, we discuss internal aspects of the VM. We explain how particular types of instructions are handled by the VM. Also, how LLVM influenced in the design of the VM.

3.3.1 What is a handler?

A handler is a function of the VM and it is invoked in order to process an instruction. The handler updates the state of the VM accordingly to the semantic of the processed instruction. Thus, there is one handler per each bitcode type.

The tasks a handler carries out are:

- to read the instruction's operands (they are register ids)
- to load the values of the necessary registers
- to perform the particularities of the instruction type
- to increase the VM's program counter accordingly to the size of the read instruction
- to indicate if the execution of the VM must continue

An example

In Listing 3.1 there is a high level representation of an add instruction handler. This procedure is executed every time an add instruction must be processed. In other words, when the virtual machine's program counter points to an instruction of this type.

When a program is obfuscated the handler and the rest of the virtual machine is created programatically using LLVM. That's why we show a high level example. The actual code is generated as LLVM-IR instructions.

```
bool add_handler(Instruction ins, Registers regs){
    register_id reg_id_0 = ins.get_operand(0);
    register_id reg_id_1 = ins.get_operand(1);
    register_id reg_id_2 = ins.get_operand(2);

    uint64_t register_value_0 = rs.load_reg(reg_id_0);
    uint64_t register_value_1 = rs.load_reg(reg_id_1);

    uint64_t computed_value = register_value_0 + register_value_1;
    regs.store_register(reg_id_2, computed_value);

    program_counter += sizeof(register_id)*3;

    // add handler does not finish the execution
    // vm's cycle must continue until a return is found
    return true;
17 }
```

Listing 3.1: High level representation of the add handler

- In lines 2, 3 and 4, we read the instruction's operands (they are register ids)
- In lines 6 and 7, we load the values of the necessary registers
- In lines 9 and 10, we perform the particularities of the instruction type
- In line 12, we increase the VM's program counter accordingly to the size of the read instruction
- In line 16, we indicate if the execution of the VM must continue

3.3.2 Call instructions

A call instruction has as operands:

- a pointer to a function
- each function's parameter as an operand

For sake of simplicity, we decided to create one call bitcode type per each signature spotted during the translation step. In this way, we do not require any signature information or the number of arguments. The bitcode type is already associated to a signature during the translation step.

Example

Consider that in our program we have two functions with the following signature:

- void foo(int a)
- void bar(bool a)

Then, we have a function F that is being obfuscated by the virtual machine. In it, we can find two calls one to foo and another to bar.

In F's bitcodes, we find two instructions corresponding to the calls to foo and bar. They both are different bitcode's type. In Table 3.6, we have the VM's bitcodes for the function calls. Although there are two calls, they are translated to two different bitcode types (call$_{int}$ and call$_{bool}$).

bitcode type	register id (function pointer)	register id (operand)
call$_{int}$	r$_i$	r$_j$
call$_{bool}$	r$_k$	r$_l$

Tab. 3.6: Bitcodes for call instructions

3.3.3 Load & Store instructions

A load instruction's operands are:

- a register id for the loaded value
- a register id for the address to be read
- a register id for the size of bytes to be read

The last operand allows us to have only one handler to manage every possible load.

The handler is easily implemented by using the memcpy function from the C standard library. Instead, we could have chosen to implement it as a loop.

In contrast, if there were no field for the size of bytes, we would be forced to have one load bitcode for fixed sizes. They could be more efficient since they have a direct correspondence in LLVM and possibly they can be better optimized. Although concerning development, it requires more time to be implemented.

Same reasoning and decisions are behind the store bitcode.

3.3.4 Ret instructions

The ret bitcode has one operand:

- the register id where the result is

If the obfuscated function is void, the instruction operand should be 0. We are forced to place a register id even if it is void because our instruction set must have a fixed number of operands. Alternatively, we could have implemented two types of ret (one with a result and one without it).

3.3.5 Size sensitive instructions

The VM is only capable of handling integers. The implementation for most instructions is the same regardless of the signess of the operands. When this is not the case, two versions of the instruction are provided (signed/unsigned operand implementations).

As we previously explained, every register id corresponds to a register. The register always has 64 bits allocated for its value. Therefore, it can store a smaller value than its maximum capacity. If we always use the 64 bits to compute an instruction, we could end up with a wrong result. Consider the next concrete example.

Example

Consider the case where the VM obfuscates the pseudo-code in Listing 3.2. The pseudo-code does a signed division between two 1-byte variables. In the comments, we can see the two complement representation of the constants (considering they are stored in 1-byte variables). The last comment is the bitcode the VM processes in order to perform the shown division. The first operand is $r0$ and it is where the result is going to be stored. The second operand is $r1$ and represents the numerator of the division, while the third operand $r2$ is the divisor. The last operand $r3$ indicates how many bits in $r1$ and $r2$ must be cleaned. All registers represent a 64 bits value, but they could be storing a smaller value. In this example, $r1$ and $r2$ are storing 1-byte constant and there is an unused space of 7 bytes in each register.

```
// signed division executed in the VM

int8_t A = 30; // 00 01 11 10
int8_t B = −5; // 11 11 10 11

int8_t C = A / B;

// bitcode: sdiv r0, r1, r2, r3
```

Listing 3.2: Division to be executed in the VM

In Table 3.7 we have the state of the registers before processing *sdiv* bitcode.

register id	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
r_1	x_7	x_6	x_5	x_4	x_3	x_2	x_1	00011110_2
r_2	y_7	y_6	y_5	y_4	y_3	y_2	y_1	11111011_2

Tab. 3.7: State of the registers just before processing the division

If the VM performs the division between r_1 and r_2, it uses the 64 bits in each register. We cannot make any assumption about the bits that are not meant to be part of the calculation. The reuse of registers during one execution of the VM can generate trash bits. A 1_2 or 0_2 as the highest bit of r_1 or r_2 can make a misleading interpretation of the sign and thus computing an incorrect value. The VM must perform the division only between the least eight significant bits in r_1 and r_2.

That is when r_3 comes into play. It is used to sign extend the specific bits in order to compute the division correctly. The values used for the arithmetic operation are the ones in Table 3.8.

The values in r_1 and r_2 remain unchanged after the execution of the bitcode. They still are the ones in Table 3.7.

The result of the division is stored in r_0. At the end the division is performed between 64 bits values. In this way, the size of the result has the same size.

register id	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
r_1	$0x00_{16}$	$0x00_{16}$	$0x00_{16}$	$0x00_{16}$	$0x00_{16}$	$0x00_{16}$	$0x00_{16}$	00011110_2
r_2	$0xFF_{16}$	$0xFF_{16}$	$0xFF_{16}$	$0xFF_{16}$	$0xFF_{16}$	$0xFF_{16}$	$0xFF_{16}$	11111011_2

Tab. 3.8: Signed extended values to 64 bits for the division

3.4 The obfuscated function

Here, we explain how a function is transformed after the obfuscation is applied. The transformation erases the original body of the function and creates a new one such that:

- has a buffer with the bitcodes to be processed by the virtual machine
- has an array with pointers to the virtual machine's handlers functions
- allocates and initializes the initial value for the virtual machine's registers
- a call to the execution loop of the interpreter

In this way, if an attacker tries to reverse engineer the binary he must understand how the virtual machine works. The original body of the function is completely removed.

Example

The new body is built using LLVM. There is no high level code involved.

The Listings 3.3 and 3.4 show the transformation's changes in a pseudo-code perspective. The first one is a function to be obfuscated while the second one is the result of the obfuscation.

```
void top_secret(){
        uint8_t* p = 0xABCD;
        uint8_t a = 50;
        uint8_t b = 25;

        uint8_t sum = a + b;
        *p = sum;
}
```

Listing 3.3: Function to be virtualized

```
void obfuscated_top_secret(){

    // alloc space for 4 registers
        Registers reg = AllocRegisterSpace(4);
        // set initial context of the VM
        reg.set_register(0, 50);
        reg.set_register(1, 25);
        reg.set_register(3, 0x0..);

        Handler handlers = {add_handler, store_handler};
        bitcode bitcodes = VM_INSTRUCTIONS_AS_BYTES;

        execution_cycle(reg, handlers, bitcodes);
}
```

Listing 3.4: Function after obfuscation

3.4.1 What is the initial context of the registers?

The initial context are the values not computed by the VM. Those are values that must be present at the start of the execution. A list of values that are considered part of the initial context is next:

- Constants
- Function's arguments

The translation step is where LLVM-IR instructions are translated into bitcodes of the virtual machine. It is there when we track which constants or arguments are used. Once this step is finished, we have identified all values of the initial context.

In LLVM-IR the stack space is modeled by a special instruction (alloca). That instruction returns a pointer of a specific type. The address corresponds to a position in the current stack frame. There is no counterpart for it in the virtual machine. Those instructions are not virtualized and they are considered part of the initial context. In other words, they are executed before the virtual machine starts and its results are assigned to specific registers.

3.5 Implementation details

3.5.1 How to mark a function to be obfuscated

To identify the functions to be obfuscated, we marked them by assigning them to a special code section. The attribute is persistent at the LLVM-IR. Listing 3.5 shows an example of how it is used.

```
#define OBFUSCATE __attribute__((section("VM")))
void OBFUSCATE foo(){
        // ...
}
```

Listing 3.5: Attributes are used to indicate which function obfuscate

3.5.2 Bitcode generation from LLVM-IR

LLVM provides a Visitor class to traverse the instructions in a function. There is one function per instruction type that can be overriden. There we write the translation logic for each instruction type.

How do we translate the operands in a LLVM instruction to a virtual machine's register? The operand in a LLVM instruction is called LLVM Value and it can be a register or a constant. LLVM instructions are in single static assignment form, that guarantees that one register is only assigned the result of one instruction. We implemented a Register Allocator that maps LLVM Values to VM's registers. Our implementation uses a naive algorithm in which there is one to one mapping between LLVM Values and VM's registers. The major drawback of this decision is that there can be a high amount of LLVM registers because they are in single static assignment form. We leave as future work to implement a better Register Allocator algorithm.

Example

In Listing 3.6 we have the translation routine for the add instruction.

1. Each LLVM Value gets a VM register id.
2. The VM instruction is created with the assigned register ids.

```
void visitAdd(BinaryOperator &I) {
  register_id reg = LLVMValueToVMReg(&I); // result of the operation
  register_id op0 = LLVMValueToVMReg(I.getOperand(0));
  register_id op1 = LLVMValueToVMReg(I.getOperand(1));

  Inst& inst = vmBasicBlock->CreateInstruction(ADD);

  inst.AddOperand(reg);
  inst.AddOperand(op0);
  inst.AddOperand(op1);
}
```

Listing 3.6: Translation of a LLVM instruction to a VM bitcode

3.5.3 Building VM handlers using LLVM

We already introduced VM handlers. Here, we describe how they are built during compilation time.

All handlers share a common behaviour. The task they perform can be summarize as follows:

1. Read the register ids in the instruction being processed.
2. Load the needed values from the register ids.
3. Perform the handler specific computations.
4. Update the value of the virtual program counter.

In order to avoid code repetition, we design a hierarchy of classes. The base class is called HandlerBuilder, it has the logic to build the necessary code for the tasks shared between different handlers. Then, each specialization only needs to implement the specific task they must perform.

Example

Listing 3.7 is an example of one HandlerBuilder. Remember that the snippet represents a way to build code by using LLVM. That is, the shown code is not the handler itself. Instead, it is the builder that during compilation time generates the intended handler.

```
void BinaryOperatorHandlerBuilder::HandlerSpecific(){
    Value* registerId0 = insParser->GetOperand(0);
    Value* registerValue1 =
        insParser->GetValueInRegisterId(insParser->GetOperand(1));
    Value* registerValue2 =
        insParser->GetValueInRegisterId(insParser->GetOperand(2));
    Value* binop = binop =
        builder->CreateAdd(registerValue1, registerValue2);
    registerStoreBuilder->set(registerId0, binop);
    readOperands = 3;
}
```

Listing 3.7: Example of a handler builder

3.6 A parametric obfuscation

If an attacker success to understand how the virtual machine works, the whole protection becomes useless. That's why our obfuscation is parametric. The goal is to generate a different virtual machine each time the obfuscation is applied. Therefore, an attacker's previous knowledge becomes useless. We consider this approach is mainly useful against manual reverse engineering.

3.6.1 Opcodes randomization

Every VM bitcode has an id that indicates the type of instruction. It is used to select the correct handler and then process it.

The idea is to randomize those ids but keeping the same number and type of bitcodes. In this way, if an attacker previously reverse engineered the semantic meaning of each bitcode id, he or she will have to re-do all the work.

3.6.2 Instruction merging

Every bitcode has its handler, with the logic to process it correctly. The VM instruction set architecture (ISA) is composed by bitcodes each having a straightforward meaning.

This option aims to:

- change the number of operations
- create a new bitcode with a non-straightforward meaning

It is achieved by combining consecutive VM bitcodes into a new one. The choice of which bitcode pattern combine is arbitrary. Thus, leading to a not clear meaning for the new bitcode.

The new bitcode's operands are the operands of the former bitcodes. The total number of new operands is equal to the sum of all the previous operands.

A new handler is automatically created based on each particular handler of every bitcode in the pattern to be merged.

When the new bitcode is processed, it updates the VM's state in the same way as it is processing all bitcodes of the pattern individually.

4. CHALLENGING THE VM OBFUSCATION

As part of our work, we prepared a challenge to face real reverse engineers. In this chapter, we describe it and explain each type of attack and their level of success. The results were crucial to show vulnerabilities and countermeasure them.

Listing 4.1 is a simplified version of it. The challenge is composed by:

- A non cryptographically-strong hash function (it can be easily inversed): MurmurHash32
- An arbitrary selected key (seed)
- A magic number, the result of applying the hash on the selected key.

The objective for the reverse engineers was to find any key that leads to the same magic number. In other words, to find an input that makes the function evaluate to true. The hash function is inlined and the whole function in Listing **??** runs inside the VM.

```
#define MAGIC_NUMBER 0123456789

bool OBFUSCATE level0(char const* key){

  if (hash(key) == MAGIC_NUMBER)
      return true;

  return false;
}
```

Listing 4.1: Challenge's pseudo-code

The challenge had a time limit of 8 hours. Three reverse engineers participated and three types of attacks are performed. The attacks are summarized in Table 4.1.

Attack	Tool	Result
Traditional attack	IDA Disassembler	Challenge unresolved. Only identifies parts of the VM.
Alternative attack #1	Triton's analysis: DSE & Taint Analysis	Challenge solved. Finds one key.
Alternative attack #2	Devirtualization technique	Generates a new binary without the VM obfuscation.

Tab. 4.1: Summary of attacks

4.1 Traditional attack

Fig. 4.1: IDA Disassembler

Two reverse engineers participated in the challenge. They used IDA Disassembler to inspect and to debug the binary. They couldn't solve the challenge. Although, they could identify parts of the VM.

4.2 Alternative attack #1

Fig. 4.2: Triton: A dynamic binary analysis framework

The attacker used Triton in order to build a script. The approach consisted in:

1. Running the challenge binary with a random input.
2. Tainting the input and manually analyzing the trace of tainted instructions.
3. Finding an assembly cmp instruction close to arithmetic instructions.
4. Retrieving a logical formula of the hash computation in terms of the input (using Dynamic Symbolic Execution).
5. Querying an SMT solver to find instantiations of the input in the formula such that the hash's formula equals the magic number.

The attacker correctly assumed that the virtual machine for any input at some execution point compared the computed hash against the magic number. Also, he correctly assumed that the comparison was ultimately done by an assembly cmp instruction.

The benefit of doing Dynamic Taint Analysis was to filter out all assembly instructions not having as an operand the input or a value dependent of it. In this way, we could remove the machinery of the virtual machine. The assembly cmp instruction was tainted because it had as an operand the hash that was computed by using the input.

If the attacker can retrieve a formula of the computed hash with the user input as a variable, he can use an SMT solver to find instantiations of the variable such that the hash equals the magic number. Logically speaking, the problem is reduced to find an instantiation of x such that hash(x) equals the magic number.

The attacker did do all of the above by using the Dynamic Symbolic Execution provided by Triton.

In this way, the attacker solved the entire challenge without having to deal with the virtual machine's implementation.

4.3 Alternative attack #2

The attacker used a devirtualization technique[1]. It is explained in Section 2.4.2. He successfully managed to create a new binary that no longer had the virtual machine obfuscation. The new binary behaved the same regarding inputs and outputs.

In [1] the steps are summarized as follows:

1. Identify input.
2. On a trace, isolate pertinent instructions using a dynamic taint analysis.
3. Build a symbolic representation of these tainted instructions.
4. Perform a path coverage analysis to reach new tainted paths.
5. Reconstruct a program from the resulting traces and compile it to obtain a devirtualized version of the original code.

Step 0 was the only one done manually. Every other step did not require any human intervention.

The implementation of the technique can be found in [10]. It is built on top of Triton and other libraries. This attack is based on the fact that the virtual machine obfuscation is vulnerable to Taint Analysis and Symbolic Execution.

4.4 Conclusions

The frustrated manual attempt (the traditional attack) of reverse engineering validated our hypothesis. That is, the virtual machine obfuscation is resilient against manual attacks.

The alternative attacks #1 and #2 are examples of the flaws in our obfuscation. The obfuscation shows no resilience against dynamic program analysis such as taint analysis and dynamic symbolic execution.

Also, they expose the need of previously obfuscate the program to be virtualized because it is possible to remove the VM obfuscation. In this way, even after the devirtualization, the code is still protected.

In the next chapters, we present complementary obfuscations to countermeasure dynamic program analysis.

5. THE LOOKUP TABLES OBFUSCATION

In this chapter, we present one of the two developed countermeasures against dynamic symbolic execution. It has been particularly tested against Triton's DSE. However, [11] claims that our reasoning works against a wider range of tools.

The obfuscation replaces instructions with memory load instructions, taking advantage of the DSE memory modelling. The resulting symbolic representation of the program becomes useless for reasoning.

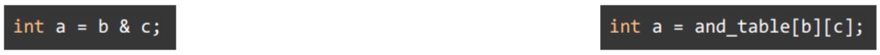
The lookup table obfuscation is implemented independently of the VM obfuscation. The idea is first to protect the code with this obfuscation and then virtualize it. In this way, we get the best of both worlds. Symbolic execution on a virtualized binary is no longer possible and the attacker requires to manually intervene.

5.1 Lookup tables

We call a lookup table an array generated during compilation time such that:

- it encodes a particular instruction (later we extend the concept to set of instructions)
- if it is indexed by the instruction's parameters, it returns the result of the instruction

5.1.1 A high level example of the transformation



```
int a = b & c;
```

```
int a = and_table[b][c];
```

Fig. 5.1: On the left is the non obfuscated code and on the right the obfuscated one

In Figure 5.1 we have the result of our obfuscation. At execution time, the ***and*** instruction is no longer visible. Instead, a memory load is performed. The ***and_table*** array is generated during compilation time and embedded into the binary.

5.2 Advantages

The advantages of this obfuscation are:

- hiding the obfuscated instruction behind a load access.
- depending on the scenario, the reverse engineer may need to understand how the constants are generated.

- DSE does not generates a faithful representation of the obfuscated instruction (explained later).

5.3 Drawbacks

The drawbacks on the obfuscation are two:

- the size of the generated array
- an easy to understand relationship between the array's constants and the parameters used to index it.

5.3.1 Size of the generated array

As you may expect, the size grows exponentially in terms of the instruction's parameter size. In the following table, we calculate the size for the ***and_table*** in terms of parameter's sizes.

operand's size (bits)	array's size
32	36893488 terabytes
16	4,3 gigabytes
8	32,8 kilobytes
4	64 bytes
2	2 bytes

Tab. 5.1: Size of the array for different parameter's sizes

In practical terms, the array for 8 byte operands or lower is feasible.

5.3.2 Straightforward constants

The second issue is the resilience of the obfuscation. For example, the array generated in Figure 5.1 could be considered trivial. The ***and_table*** can be easily understood because it is a very simple operation.

5.4 Addressing unfeasible lookup tables in terms of size

In the context of our work, we only obfuscate no carry operations. Specifically, ***and***, ***xor*** and ***or*** operations.

Assume we have an operation op to be obfuscated. Consider a and b the operands and c the result. Also, consider that a, b and c have the same size and it is a power of two. We denote the size of the operands and the result as $size$.

We can use multiple times the same operation but for smaller operands to compute the original *op*. The operation is op_s with operands a_s, b_s. The result is c_s. We denote the size of the operands as $size_s$ and it is a divisor of $size$.

The idea is to take the first $size_s$ bits of a and b, and perform the result with op_s. In this way, we get the first $size_s$ bits of c. Then, we proceed to take the next $size_s$ bits of a and b. We compute op_s between them, and we get the next $size_s$ bits of c. The process is repeated until we get all bits of c by calculating multiple times op_s with chunks of a and b.

Finally, we replace every use of op_s by an access to the lookup table of op_s. In this way, we are using a smaller table to compute the operation between bigger operands.

5.4.1 A high level example

In the example, we compute the 8-bit version of the ***and*** with a lookup table for 4-bit operands. Similarly, the example could be about computing the 32-bit version of the ***and***.

In Figure 5.1 we have the instruction to obfuscate. It is the ***and*** operation between operands ***b*** and ***c***.

In Figure 5.2 we show how to compute the original ***and*** using chunks of 4-bit values.

Finally, in Figure 5.3 we replace uses of the ***and*** with a lookup table indexing.

```
void foo(uint8_t b, uint8_t c){
        // ...
        uint8_t a = b & c; // instruction to obfuscate
        // ...
}
```

Listing 5.1: Original code

```
void foo(uint8_t b, uint8_t c){
        // ...

        uint8_t a0 = get_first_four_bits(b) & get_first_four_bits(c);
        uint8_t a1 = get_second_four_bits(b) & get_second_four_bits(c);

        uint8_t a = (a1 << 4) | a0;

        // ...
}
```

Listing 5.2: Instruction replaced by two uses of the same operation between smaller operands

```
void foo(uint8_t b, uint8_t c){
        // ...

        uint8_t a0 = and_table_4_bits[get_first_four_bits(b)]
```

```
                                     [get_first_four_bits(c)];
      uint8_t a1 = and_table_4_bits[get_second_four_bits(b)]
                                     [get_second_four_bits(c)];

      uint8_t a = (a1 << 4) | a0;

      // ...

      // note: and_table_4_bits is expected
      // to be only indexed by integers
      // in the domain of 4 bits integers
      // (although they are stored in a byte)
}
```

Listing 5.3: Replace operations between smaller operands with multiple lookup table accesses

5.5 Strengthening the constants

In this section, we explain how to create harder to reverse lookup tables. The idea is to increase the effort to deduce which operations build those constants. Again, we restrict our scope to no carry operations. Specifically, ***and***, ***xor*** and ***or*** operations.

5.5.1 User chains

In LLVM, the *users* of an instruction ***i*** are all other instructions that *use* the result of ***i*** as an operand.

We generalize that concept and define the *user chain* term. Intuitively, they are instructions (computations) that *use* the result of another instruction belonging to the chain as an operand

More precisely, a *user chain* is a set of operations such that:

1. It has a only one *initial* operation. Neither of the *initial* operation's operands are a result of another operation in the *user chain.*
2. Every other operation in the *user chain* has at least one operand computed by another operation in the *user chain.*

Additionally, we call *user chain's parameters* a set of operands such that:

1. The operands are not computed by another operation in the *user chain.*

Lastly, we call *user chain's final operation* one operation such that:

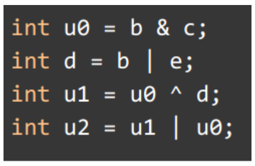
1. It belongs to the *user chain.*
2. The result of it is not used by any other operation in the *user chain.*

Therefore, we can conclude that fixing the *user chain's parameters* with concrete values we determine the value of the *user chain's final operation.*

The operands not present in the *user chain's parameters* are a result of another operation in the *user chain*. The operands in the *user chain's parameters* are assigned a fixed value.

A high level example

User chains are considered at LLVM-IR level. However, we present a high level perspective of one of them. Remember that LLVM-IR is in SSA form. In the example, we mimic an SSA code.



```
int u0 = b & c;
int d = b | e;
int u1 = u0 ^ d;
int u2 = u1 | u0;
```

Fig. 5.2: Example of a user chain

The *user chain* is composed of the operations $u0$, $u1$ and $u2$. Consider that:

- $u0$ is the only *initial* operation
  - its operands are not in the *user chain*
  - $u0$ is used as an operand in the *user chain*
- $u1$ and $u2$ each have at least one of their operands in the *user chain*
- $u2$ is the *final* operation because it is not used as an operand in the *user chain*
- b, c and d are *parameters* of the user chain because they are not calculated by any operation in the *user chain*

5.5.2 Folding a user chain into a lookup table

In the scope of this work we only consider *user chains* made of ***and***, ***xor*** and ***or*** operations.

The idea is to create a lookup table of a reasonable size given a *user chain*. Then, replace all uses of the *user chain's final operation* with indexes into the lookup tables (as explained in 5.3.2). The indexing is done with the *user chain's parameters* because they determine the value of the *final* operation (explained in Section 5.5.1). Lastly, we remove the *user chain* from the code. If any of the *user chain's* operations are used as an operand, we copy and insert them just before each use.

The size of the lookup table associated to it is expressed in terms of:

- the number of parameters: $params$
- the size of a parameter in bits: $size(param)$ (they all have the same size)
- the number of values in a parameter: $2^{size(param)}$

- the number of different parameter combinations: $params * 2^{size(param)}$
- the allocated space in bits for each array cell: $alloc$ (it equals $size(params)$ because there is no carry)

Then, the lookup table's size equals: $\frac{params*2^{size(param)}*alloc}{2}$

Because of the symmetry property of bitwise operations, we can reduce the size of the table by storing only half of it. Also, consider that an *alloc* value not multiple of 8 bits creates constants not aligned in memory. That would require more operations to fetch each value.

An example

Consider the *user chain* in 5.2. The *user chain* uses 32-bit values. However, we know we can use the 4-bit version to compute it. For the *alloc* value, we use 1 byte. In this way, our constants are aligned in memory.

The lookup table's size for the 4-bit version is computed in the following way:

- the number of parameters: $params = 3$
- the size of a parameter in bits: $size(param) = 4\ bits$ (they all have the same size)
- the number of values in a parameter: $2^{size(param)} = 16$
- the number of different parameter combinations: $params * 2^{size(param)} = 64$
- the allocated space in bits for each array cell: $alloc = 8bits$ (it equals $size(params)$ because there is no carry)

Then, the lookup table's size equals: $params * 2^{size(param)} * alloc = 3 * 16 * 8 = 192$ bits $= 24$ bytes. However, since the table is symmetric, we only really need half of it.

A partial view of the lookup table is in 5.2. The indexes are b, c and d and the only value stored in the array is $u2$.

indexes/parameters			
b	c	d	u2
0x09	0x07	0x09	0x09
0x09	0x07	0x0a	0x0b
0x09	0x07	0x0b	0x0b
0x09	0x07	0x0c	0x0d
0x09	0x07	0x0d	0x0d

Tab. 5.2: Partial view of the lookup table (generated at compilation time)

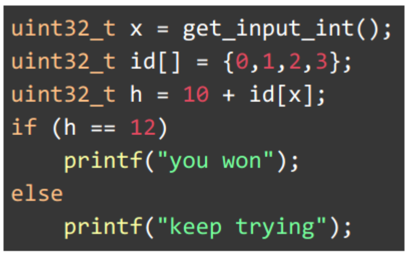
6. CHALLENGING THE LOOKUP TABLES OBFUSCATION

In this section, we set up two experiments. The first one illustrates how Triton's symbolic expressions are not enough to reason about the target program [11]. Also, we consider possible workarounds an attacker could use. Even though such workarounds exist, they are non-scalable.

The second one tests the resilience against the devirtualization method [1] over a set of hash functions. The implementation of the technique is on top of Triton's framework.

6.1 Limitations of symbolic expressions

Consider the code in Figure 6.1. We want Triton to find a possible input to reach the winning message. The input to print the winning message is the number two.



```
uint32_t x = get_input_int();
uint32_t id[] = {0,1,2,3};
uint32_t h = 10 + id[x];
if (h == 12)
    printf("you won");
else
    printf("keep trying");
```

Fig. 6.1: Example of how to have a misleading symbolic expression

Note that the array is indexed with the user's input. In Dynamic Symbolic Execution, the input is considered a Symbolic Variable. The Symbolic Variable is used to build formulas resembling how concrete values of the current execution path are logically built. A formula holding a Symbolic Variable is only valid for all concrete inputs sharing the taken path constraints up to that point. Then, a SMT solver can reason with the formula and finding possible concrete values for the Symbolic Variable to make a query hold true.

We run the example with an initial seed (ie: one) and Triton runs at the same time performing DSE. At the assembly address where h is computed, we ask Triton to get the symbolic expression of h. Then, we make a formula such that h's symbolic expression equals twelve. Triton's answer is that there is no possible solution. However, we know that to be false.

The symbolic expression built for h is the one in Figure 6.2. In it, we can see that the formulas are building a constant value. At any given point, there is no Symbolic Variable involved. This behaviour is a consequence of two implementation characteristics of Triton's DSE:

1. If the loaded value from memory was not calculated during the execution of Triton's DSE, then the symbolic expression for it is a constant symbolic expression.

The constant in the array is not computed during the execution, therefore Triton associates a constant symbolic expression for each value in the array.

2. Triton only builds formulas using integer logic and a memory address can only be linked to a single symbolic expression at any point of the execution.

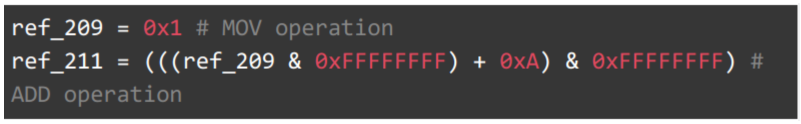
This point limits the behaviour of Triton when loading an address calculated in terms of the input, like the one in our current example. Triton searches the symbolic expression linked to the concrete address being loaded, and links the destination address or register to the retrieved symbolic expression.

In the case of h, this behaviour links it only to one symbolic expression because of point (2) and to a constant because of point (1).

Basically, Triton maps h to the symbolic expression referenced from the concrete address of $id[1]$ (in our running example, x is one).

Lookup tables introduce the same issues if they are indexed by values computed in terms of the input. An indexation into them corresponds to a input dependant memory load.

Therefore, lookup tables exploit point (1) because they are calculated in compilation time. Also, they exploit point (2) if they are indexed by values related to the input.



```
ref_209 = 0x1 # MOV operation
ref_211 = (((ref_209 & 0xFFFFFFFF) + 0xA) & 0xFFFFFFFF) #
ADD operation
```

Fig. 6.2: Symbolic expression for h (concrete input: 1)

6.1.1 Workaround in the devirtualization implementation

Recall that [1] implementation (available in [10]) is built on top of Triton and it uses its Dynamic Symbolic Execution. In Section 6.1, we described the limitations that DSE can face against input dependant memory accesses.

[10] is aware of Triton's limitations and implements a workaround. Although, it is not enough. Briefly, it can be summarized like this:

- Before processing a memory access it checks if the address is in terms of the input (a.k.a symbolized)
- If it is symbolized, then it issues a query to the SMT solver. The query asks for:
  - An input that drives the execution to the exact same point.
  - Generates a different concrete address than the current one.

The main issue of this approach is that it does not constraint the size of the input. Thus, leading into a non scalable number of inputs as solutions for the query.

The fundamental aspect of the technique [1] is to generate inputs to explore each path of the program. Using DSE it implements concolic execution. An exponential number of inputs to explore new paths can make the tool timeout.

6.2 Effectiveness against the devirtualization technique

In this section, we report the resilience of the lookup table obfuscation against devirtualization attacks using technique [1].

6.2.1 Lookup tables obfuscation

We obfuscate the hash functions with the lookup table obfuscation. Then, we try to devirtualize the obfuscated binaries. The results are displayed in Table 6.1

hash	timeout	inputs to explore	explored inputs
adler	**yes**	16315	4
cityhash	**yes**	92542	11
fnv1a	**yes**	398343	98
jenkins	**yes**	418181	20
siphash	**yes**	8415	1
spookyhash	**yes**	185092	22
superfasthash	**yes**	348805	36
md5	**yes**	-	-

Tab. 6.1: Devirtualization technique results on hashes obfuscated with lookup tables

In all cases, the devirtualization technique timeouts. After the timeout, it reports the number of inputs pending to be explored. Also, how many of them were explored before timing out.

6.2.2 Lookup tables obfuscation and virtualization

We also test the resilience of the lookup tables and the virtual machine together. In this scenario, we first apply the lookup tables on one hash and then virtualize the whole algorithm.

In Table 6.2 we report the results. The devirtualization technique fails again and in some cases it crashes.

hash	timeout	inputs to explore	explored inputs
adler	**crash**	-	-
cityhash	**yes**	137303	91
fnv1a	**yes**	230718	233
jenkins	**crash**	-	-
siphash	**crash**	-	-
spookyhash	**yes**	200980	88
superfasthash	**crash**	-	-
md5	**crash**	-	-

Tab. 6.2: Devirtualization technique results on hashes obfuscated with lookup tables and virtualized

6.3 Performance of the obfuscation

In order to be complete, we report the performance of the hashes. In one case, we obfuscate them just with the lookup tables. In the second one, we obfuscate with the lookup table and then we virtualize it. We follow the same experiment setup as in Section 9.2.1.

6.3.1 Baseline

As a baseline, we report the running time of the binaries when they are not obfuscated. The measurements are in Table 11.6.

6.3.2 Lookup tables

The Table 11.7 in the Appendix has the average running time for the obfuscated hashes.

Here, we show a comparison between the non obfuscated binaries and the obfuscated ones. Table 6.3 expresses the overhead incurred by the obfuscation. For instance, the md5 hash when obfuscated with the lookup tables it is 1,33 times slower in average than the non obfuscated version.

	lookup tables overhead							
	obfuscated running time / non obfucasted running time							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	1,33	1,95	1,79	3,19	3,37	4,91	5,65	10,86
median	1,34	1,95	1,78	3,19	3,37	4,89	5,65	10,86
stddev	1,21	0,77	1,23	6,1	4,2	14,57	4,58	32,4

Tab. 6.3: Lookup table overhead in hash functions

7. THE RANGE DIVIDER OBFUSCATION

Range divider increases the number of possible paths in a program, such that every new path is important for preserving the semantics. Since they are feasible paths, a concolic execution on the programs requires to do more work. Mainly, because it needs to generate concrete input to explore all feasible paths of the target program. A SMT solver generates the new input. That is why this obfuscation increases the running time of the conocolic execution. Also, if the SMT solver is not able to find a solution, the exploration is incomplete. Therefore, any type of analysis becomes incomplete.

The devirtualization technique [1] uses a concolic execution in order to generate its result. Because of it, we find range divider an effective technique to countermeasure a devirtualization attempt or any tool based on a concolic execution.

7.1 The transformation

The core concept behind the obfuscation is to split the domain of a computation in our program. The steps can be summarized as:

1. Select a partial computation
2. Partition its domain
3. In each case, recompute it

In order to be effective, we require to select an input dependant computation. Also, we require it to be used as part of the final computation. Otherwise, a backward slicing algorithm could remove the protection.

In the context of this work, we partition the domain by using the modulo operation with a fixed constant. In each case of the modulo, we re compute the value. However, this idea is generalizable to any way of partition the domain of a value.

It is important to obfuscate again each case of the modulo. We make an attacker think we are not just recomputing a value.

7.2 A high level example

Here, we show a high level example of the obfuscation. In the Listing 7.1 we have the original function. After obfuscating, the code is the one in Listing 7.2. The code is presented in C/C++. However, the obfuscation is implemented at LLVM-IR level.

```
uint32_t add(uint32_t x, uint32_t y) {
  uint32_t res = x + y;
  return res;
}
```

Listing 7.1: Non obfuscated function

```
uint32_t add(uint32_t x, uint32_t y){
    uint32_t res = 0;

    uint32_t c = 3;
    uint32_t r = x % c;

    // x mod c = r <=> exists m such that x = c*m+r

    if (r == 0){
        uint32_t m_recomputed = (x-0)/c;
        uint32_t x_recomputed = m_recomputed*c + 0;
        res = x_recomputed + y;
    } else if (r == 1){
        uint32_t m_recomputed = (x-1)/c;
        uint32_t x_recomputed = m_recomputed*c + 1;
        res = x_recomputed + y;
    } else if (r == 2){
        uint32_t m_recomputed = (x-2)/c;
        uint32_t x_recomputed = m_recomputed*c + 2;
        res = x_recomputed + y;
    }

    return res;
}
```

Listing 7.2: Function obfuscated with range divider

In Listing 7.2, we replace the original addition between x and y with a chain of ifs depending on the value of $x\%c$. The domain of x is divided in three cases, each corresponding to every possible value of $x\%c$. In every case, we recompute x using the definition of the remainder. Then, the intended addition is performed between x and y. It is important to note that the transformation is not removed by the compiler.

7.3 Nested application

One advantage that this obfuscation provides is the fact that it can be nested.

In Listing 7.2 we could apply again the obfuscation on the operand y in each case. Following this logic, the number of paths is increased exponentially.

Similarly, if we decide to transform a computation inside a loop we can have a similar effect.

The drawback is a bigger executable however, the addition is still done in $O(1)$. But the advantage outweighs the disadvantage. An attacker requires to issue more queries to an

SMT solver, which is by far more expensive. The number of queries is at least the number of new feasible paths.

8. CHALLENGING THE RANGE DIVIDER OBFUSCATION

In this chapter, we setup two experiments. The first one shows the effectiveness of our obfuscation against the devirtualization technique. The second one shows the performance penalty of the obfuscation.

8.1 Effectiveness against the devirtualization technique

In this section, we report the resilience of the range divider obfuscation against devirtualization attacks using technique [1].

8.1.1 Range divider obfuscation

hash	timeout	inputs to explore	explored inputs
adler	yes	132878	272
cityhash	yes	37092	49
fnv1a	yes	110065	245
jenkins	yes	124591	258
siphash	yes	27153	54
spookyhash	yes	28715	57
superfasthash	yes	123581	249
md5	yes	129912	266

Tab. 8.1: Range divider's resilience against devirtualization

8.1.2 Range divider obfuscation and virtualization

hash	timeout	inputs to explore	explored inputs
adler	yes	116262	236
cityhash	yes	35594	47
fnv1a	yes	108119	240
jenkins	yes	111772	233
siphash	yes	25663	51
spookyhash	yes	28715	57
superfasthash	yes	113150	235
md5	yes	-	-

Tab. 8.2: Range divider and virtualization resilience against devirtualization

8.2 Performance of the obfuscation

In order to be complete, we report the performance of the hashes. In one case, we obfuscate them just with the range divider. In the second one, we obfuscate with the range divider and then we virtualize it. We follow the same experiment setup as in Section 9.2.1.

8.2.1 Baseline

As a baseline, we report the running time of the binaries when they are not obfuscated. The measurements are in Table 11.9.

8.2.2 Range Dividers

In the Appendix, the Table 11.8 has the average running time for the obfuscated hashes.

Here, we show a comparison between the non obfuscated binaries and the obfuscated ones. The Table 8.3 expresses the overhead incurred by the obfuscation. For instance, the md5 hash when obfuscated with the range divider it is 2,27 times slower in average than the non-obfuscated version.

	range divider overhead							
	obfuscated running time / non obfuscated running time							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	2,27	1,86	2,04	1,94	2,88	3,59	2,90	4,00
median	2,27	1,85	2,03	1,96	2,91	3,56	2,91	4,00
stddev	0,88	2,05	1,62	0,97	2,41	4,52	0,27	0,48

Tab. 8.3: Range divider overhead in hash functions

9. THE VIRTUAL MACHINE'S PERFORMANCE IN CONTEXT

We expect the obfuscation to increase the execution time. In the development of this work, the performance is not an objective to achieve. However, in order to be complete, we report the obfuscation performance in different scenarios.

As we stated before, the obfuscation is not meant to be applied in an entire program. The programmer must carefully select which section of code is convenient to obfuscate. He or she has to keep in mind the performance and security trade-offs.

9.1 An image encoder

In this scenario, we obfuscate one part of an image compressor. We consider it a real scenario because it could be a proprietary compression algorithm to protect.

The project is called imagezero [12]. It's a small project in C++, whose core is a compression algorithm. In this evaluation, we obfuscate the compression routine in its entirety. However, a developer with a deeper understanding of the algorithm could reduce more the scope of the virtualization. Probably, improving the overall performance.

The function to obfuscate is called encodeImagePixels presented in Listing 11.1. Two loops traverse all pixels of the image. The function encodePixel runs on every pixel. In this benchmark, we inlined everything inside the encodeImagePixels. Lastly, we virtualize the entire code in encodeImagePixels.

9.1.1 Experiment setup

As inputs we select two images corresponding to the two possible dimensions of a 4K image. The resolutions are:

- 3840 x 2160
- 4096 x 2160

The obfuscated and non-obfuscated version of the function are run 30 times. We use the google-benchmark library to perform the microbenchmarking of the desired function. Note that the rest of the execution of the program is not part of our measurements. We report:

- average of the benchmarked time
- median of the benchmarked time
- standard deviation of benchmarked time

The results are in the appendix section in Table 11.1.

In addition, we use the perf tool. It allows us to collect information regarding CPU statistics. We are interested in the number of:

- instructions
- cycles
- branches
- branch-misses
- cache-references
- cache-misses

In the appendix section perf reports are shown in Tables 11.2 11.3 11.4.

9.1.2 Results

The obfuscation overhead is draw in Table 9.1. The table indicates the ratio between the obfuscated running times and the non obfuscated ones. For instance, the average obfuscated execution time for the 3840x2160 resolution is 97,76 times slower than the non-obfuscated.

resolution	average	median	std dev
3840x2160	97,76	97,72	1,09
4096x2160	109,91	109,84	0,65

Tab. 9.1: Virtualization overhead

The first conclusions are:

- As expected, the virtual machine overhead is high.
- In terms of absolute values, the execution time is still feasible (see Table 11.1).
- Real time (streaming) application of the algorithm becomes unfeasible.

9.1.3 A CPU perspective

	obfuscation overhead					
resolution	cache-references	cache-misses	instructions	cycles	branches	branch-misses
3840x2160	2,38	1,08	81,35	80,20	426,38	82,07
4096x2160	1,86	1,29	87,36	92,26	513,97	119,19

Tab. 9.2: Obfuscation overhead

In Table 9.2 we summarize statistics gathered by perf. A value in the table is the ratio between the obfuscated measurement and the non obfuscated measurement. For instance, cache-references in the obfuscated version is 2,38 times the non obfuscated measurement.

Based on these values we can hypothesize how the VM impacts on the CPU performance. We explain each parameter and also propose possible solutions.

cache-refrences & cache-misses

The virtual machine registers are 64 bits. Every time a handler is executed, it loads the content of the necessary virtual registers. The size of the loaded value is always 64 bits, even if the stored value is smaller. We believe this can be an overhead for the cache because we can fill faster every cache line.

A possible solution could be having different sizes for registers. In this way, there is less wasted space and therefore a more efficient use of the cache.

In addition, we have to consider branches in the program to be obfuscated. They become bitcodes of the VM and the VM handler updates an index (virtual program counter) into the bitcodes's buffer. However, this behaviour could lead to cache-misses because we could index into a neither recently accessed or near memory address.

instructions

The VM intrinsically has an impact on the number of executed instructions. The obfuscated section of code is not directly executed by the CPU. In order to execute one VM bitcode, we have to execute one time the VM execution cycle and the associated handler.

We believe this issue is hard to tackle. The number of executed instructions by the CPU can be seized by:

- reducing the number of bitcodes to execute
- reducing the code in each handler

The virtualized code is already optimized by LLVM. There is a one to one relationship between bitcodes and LLVM-IR. Therefore the only option is merging multiple LLVM-IR instructions into a single bitcode. In this way we reduce the number of bitcodes but we increase the complexity of the associated handler.

cycles & branches & branch-misses

The VM execution cycle has two assembly branches associated with it:

- a branch instruction to the handler of the decoded bitcode
- a branch to repeat the execution cycle

Every bitcode is processed by the execution cycle. We could be introducing branch-misses in order to process a bitcode and therefore having a big impact on the performance. We consider that possible solutions are similar to the ones described to reduce the number of executed instructions.

9.2 Hash algorithms

We also benchmarked the VM obfuscation on hash algorithms. The intent of the VM obfuscation is to obfuscate short sections of cryptography code. We test the following hash functions:

- md5
- adler
- jenkins
- siphash
- superfasthash
- fnv1a
- spookyhash
- cityhash

They are simple C implementations and are originally selected in [1]. As we previously stated, the scope of this work is not focused on performance. However, we report it to be complete.

9.2.1 Experiment setup

All hash implementation do not have input dependent loops. All loops are fixed to the size of the input. The size is fixed and therefore the value of the input does not change the number of executed instructions.

In the experiment, we fix the input and run each hash 30 times. Then we calculate:

- average of the benchmarked time
- median of the benchmarked time
- standard deviation of benchmarked time

We use google benchmark, it allows us to measure very short execution of code. Raw results are in the appendix 11.5. Next we draw conclusions from the results.

9.2.2 Results

The running time penalty of the obfuscation is draw in Table 9.3. It describes how many times the obfuscation is slower than the non obfuscated version. For example, md5 is 25,59 times slower on average when virtualized.

	obfuscated measurements divided non obfuscated measurements		
hash	average (times)	median (times)	std dev. (times)
md5	22,59	22,53	15,54
adler	7,83	7,83	6,38
jenkins	14,04	14,04	17,31
siphash	24,57	24,67	51,18
superfasthash	15,64	15,64	21,88
spookyhash	7,16	7,16	3,51
cityhash64	9,24	9,24	2,86
fnv1a	11,29	11,29	14,95

Tab. 9.3: obfuscated measurements diveded non obfuscated measurements

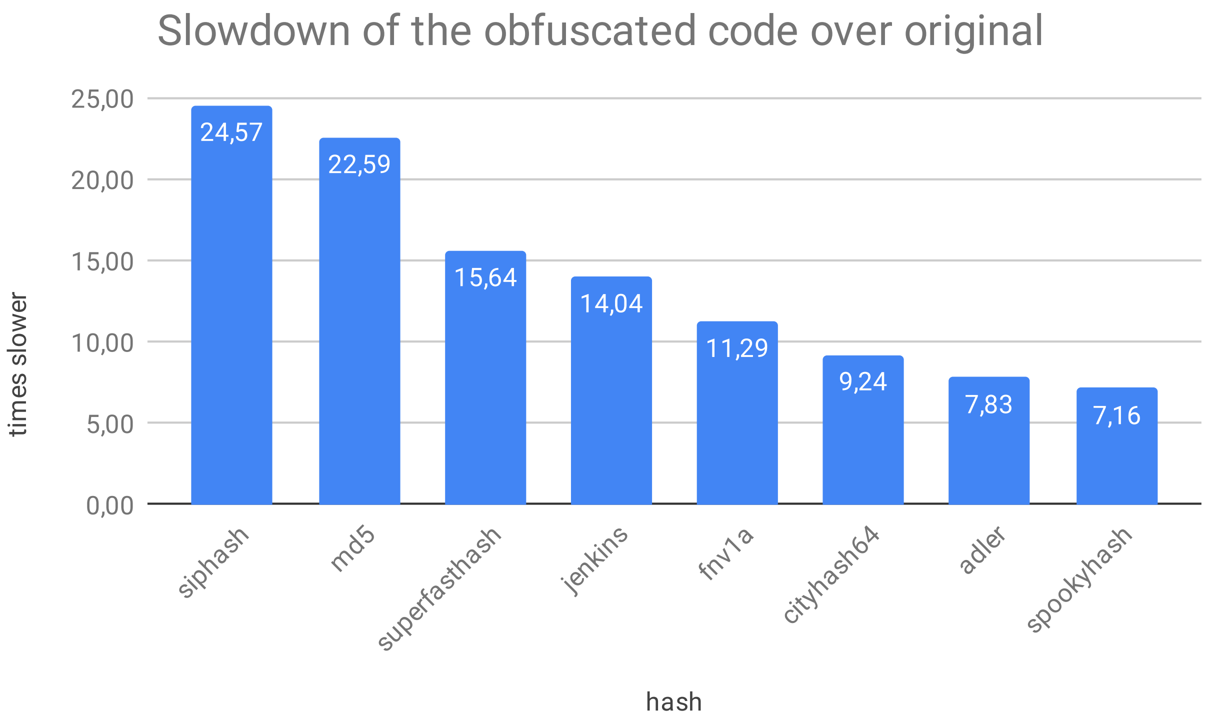


Fig. 9.1

As expected, there is a performance impact. The user of the obfuscations should choose carefully which parts of his algorithm to obfuscate. In this scenario, is not possible to run perf. The short execution time of the hashes makes it impossible to gather faithfully information about the CPU. Basically, because the tool samples CPU stats between intervals.

We leave this as a future work, to investigate the reasons of the performance penalty.

9.3 Testing correctness

In order to check the correctness of the obfuscation, we used the hash functions from the previous section. We fixed a seed and generated random inputs, then we compared the results of the obfuscated version and the non obfuscated one.

We selected 64 bits inputs for the hash functions. We choose:

- all integers between $[0, 1000)$
- 1000 random samples from [0, 0xFFFF)
- 1000 random samples from [0xFFFF, 0xFFFFFFFF)
- 1000 random samples from [0xFFFFFFFF, 0xFFFFFFFFFFFFFFFF)

In [1], the devirtualized binaries are tested in the same way and under the same set of hash functions.

10. FUTURE WORK

Next we list aspects we find interesting to keep working on:

1. Improve the performance of the VM. We could have variable register sizes. We may take better advantage of the cache.
2. Support floating point instructions in the VM.
3. Extend lookup tables and user chains to instructions with carry bit.
4. Find alternative implementations of the range divider. Currently it is only implemented considering the remainder of an integer.
5. Implement different strategies to break symbolic execution and taint analysis. For instance, we could make the data flow of a program go through a socket or file. This could possibly break the instrumentation of a dynamic analysis.

11. APPENDIX

11.1 Source of the image encoder

```
#ifndef IZ_ENCODER_H
#define IZ_ENCODER_H 1

#include <cstring>

#include "iz_p.h"

namespace IZ {

#define encodePixel(predictor)                        \
{                                                     \
    Pixel<> pix, pp;                                  \
                                                      \
    pix.readFrom(p);                                  \
    pp.predict(p, bpp, bpr, predictor::predict);      \
    pix -= pp;                                        \
    pix.forwardTransform();                           \
    p += bpp;                                         \
    pix.toUnsigned();                                 \
                                                      \
    int nl = pix.numBits();                           \
    cx = (cx << CONTEXT_BITS) + nl;                   \
    this->writeBits(dBits[cx & bitMask(2 * CONTEXT_BITS)]
        , dCount[cx & bitMask(2 * CONTEXT_BITS)]);    \
    pix.writeBits(*this, nl);                         \
}

#define OBFUSCATE __attribute__((section("VM")))

template <
    int bpp = 3,
    typename Predictor = Predictor3avgplane<>,
    typename Code = U32
>
class ImageEncoder : public BitEncoder<Code>
{
public:
    ImageEncoder() {
        memcpy(dBits, staticdBits, sizeof(dBits));
        memcpy(dCount, staticdCount, sizeof(dCount));
```

```
}

void  __attribute__((always_inline)) encodePixel_predictor0(const
            unsigned char *&p, const int bpr, unsigned int& cx){
    Pixel<> pix, pp;

    pix.readFrom(p);
    pp.predict(p, bpp, bpr, Predictor0<>::predict);
    pix -= pp;
    pix.forwardTransform();
    p += bpp;
    pix.toUnsigned();

    int nl = pix.numBits();
    cx = (cx << CONTEXT_BITS) + nl;
    this->writeBits(dBits[cx & bitMask(2 * CONTEXT_BITS)]
            , dCount[cx & bitMask(2 * CONTEXT_BITS)]);
    pix.writeBits(*this, nl);
}

void __attribute__((always_inline)) encodePixel_predictor1x(const
            unsigned char *&p, const int bpr, unsigned int& cx){
    Pixel<> pix, pp;

    pix.readFrom(p);
    pp.predict(p, bpp, bpr, Predictor1x<>::predict);
    pix -= pp;
    pix.forwardTransform();
    p += bpp;
    pix.toUnsigned();

    int nl = pix.numBits();
    cx = (cx << CONTEXT_BITS) + nl;
    this->writeBits(dBits[cx & bitMask(2 * CONTEXT_BITS)]
            , dCount[cx & bitMask(2 * CONTEXT_BITS)]);
    pix.writeBits(*this, nl);
}


void  __attribute__((always_inline)) encodePixel_predictor1y(const
            unsigned char *&p, const int bpr, unsigned int& cx){
    Pixel<> pix, pp;

    pix.readFrom(p);
    pp.predict(p, bpp, bpr, Predictor1y<>::predict);
    pix -= pp;
    pix.forwardTransform();
    p += bpp;
    pix.toUnsigned();

    int nl = pix.numBits();
    cx = (cx << CONTEXT_BITS) + nl;
    this->writeBits(dBits[cx & bitMask(2 * CONTEXT_BITS)]
            , dCount[cx & bitMask(2 * CONTEXT_BITS)]);
    pix.writeBits(*this, nl);
}

void __attribute__((always_inline)) encodePixel_predictor(const
        unsigned char *&p, const int bpr, unsigned int& cx){
    Pixel<> pix, pp;

    pix.readFrom(p);
    pp.predict(p, bpp, bpr, Predictor::predict);
    pix -= pp;
    pix.forwardTransform();
    p += bpp;
    pix.toUnsigned();

    int nl = pix.numBits();
    cx = (cx << CONTEXT_BITS) + nl;
    this->writeBits(dBits[cx & bitMask(2 * CONTEXT_BITS)]
            , dCount[cx & bitMask(2 * CONTEXT_BITS)]);
    pix.writeBits(*this, nl);
}

void OBFUSCATE __attribute__ ((noinline)) encodeImagePixels
        (const Image<> &im) {
    const int bpr = im.samplesPerLine();
    const unsigned char *p = im.data();
    int size = im.width() * im.height();
    const unsigned char *pend = p + bpp * size;
    unsigned int cx = (7 << CONTEXT_BITS) + 7;

    /* first pixel in first line */
    encodePixel_predictor0(p, bpr, cx);
    /* remaining pixels in first line */
    const unsigned char *endline = p + bpr - bpp;
    while (p != endline) {
        encodePixel_predictor1x(p, bpr, cx);
    }
    while (p != pend) {
        /* first pixel in remaining lines */
        //encodePixel(Predictor1y<>);
        encodePixel_predictor1y(p, bpr, cx);
```

```
            /* remaining pixels in remaining lines */
            const unsigned char *endline = p + bpr - bpp;
            while (p != endline) {
                encodePixel_predictor(p, bpr,cx);
            }
        }
    }

    void encodeImageSize(const Image<> &im) {
        int w = im.width() - 1;
        int h = im.height() - 1;
        int b = ::numBits(w | h);
        this->writeBits(b, 4);
        this->writeBits(w, b);
        this->writeBits(h, b);
        this->flushCache();
    }

private:
    unsigned int dBits[1 << (2 * CONTEXT_BITS)];
    unsigned int dCount[1 << (2 * CONTEXT_BITS)];
};

} // namespace IZ

#endif
```

Listing 11.1: Obfuscated source code

11.2 Image encoder results

	# iterations = 30					
	obfuscated version			non obfuscated version		
resolution	average (ms)	median (ms)	std dev (ms)	average (ms)	median (ms)	std dev (ms)
3840x2160	10460	10456	117	107	107	2,56
4096x2160	11211	11204	66,1	102	102	2,53

Tab. 11.1: Microbenchmarking results

	obfuscated version		*non obfuscated version*	
resolution	cache-references	cache-misses	cache-references	cache-misses
3840x2160	8605460	2892391	3614262	2680232
4096x2160	5891487	2771769	3154874	2145943

Tab. 11.2: Perf report: cache-references and cache-misses

	obfuscated version		*non obfuscated version*	
resolution	instructions	cycles	instructions	cycles
3840x2160	71741719611	35864779685	881881503	447183455
4096x2160	75564437028	37708451058	864986710	408708259

Tab. 11.3: Perf report: instruction and cycles

	obfuscated version		*non obfuscated version*	
resolution	branches	branch-misses	branches	branch-misses
3840x2160	14091199991	155225271	33047784	1891247
4096x2160	14832384167	162730319	28858258	1365328

Tab. 11.4: Perf report: branches and branch-misses

11.3 Hash results

	#iterations = 30					
	obfuscated binary			*non obfuscated binary*		
hash	average (ns)	median (ns)	std dev. (ns)	average (ns)	median (ns)	std dev. (ns)
md5	36896	36972	258	1633	1641	16,6
adler	517	517	1,36	66	66	0,213
jenkins	466	466	1,35	33,2	33,2	0,078
siphash	1661	1665	13	67,6	67,5	0,254
superfasthash	316	316	0,633	20,2	20,2	0,029
spookyhash	461	461	0,703	64,4	64,4	0,2
cityhash64	158	158	0,258	17,1	17,1	0,09
fnv1a	175	175	0,344	15,5	15,5	0,023

Tab. 11.5: Microbenchmarking results

11.4 Lookup tables

	#iterations = 30							
	no obfuscation							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	449	11,8	7,48	12,5	3,71	1,86	4,12	1,86
median	449	11,8	7,48	12,5	3,71	1,86	4,12	1,86
stddev	5,37	0,129	0,047	0,03	0,01	0,007	0,012	0,005

Tab. 11.6: Microbenchmarking results: baseline

	#iterations = 30							
	lookup tables							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	597	23	13,4	39,9	12,5	9,13	23,3	20,2
median	601	23	13,3	39,9	12,5	9,1	23,3	20,2
stddev	6,51	0,1	0,058	0,183	0,042	0,102	0,055	0,162

Tab. 11.7: Microbenchmarking results: lookup tables

11.5 Range divider

	#iterations = 30							
	range divider							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	901	19,9	19,9	22,3	18,3	16,1	16,8	16,9
median	900	19,8	19,8	22,3	18,3	16	16,8	16,9
stddev	4,62	0,305	0,164	0,211	0,32	0,244	0,026	0,031

Tab. 11.8: Microbenchmarking results: range divider

	#iterations = 30							
	no obfuscation							
hashes	md5	adler	jenkins	siphash	superfasthash	fnv1a	spookyhash	cityhash
mean	397	10,7	9,74	11,5	6,36	4,48	5,8	4,23
median	397	10,7	9,73	11,4	6,29	4,5	5,78	4,22
stddev	5,24	0,149	0,101	0,218	0,133	0,054	0,096	0,065

Tab. 11.9: Microbenchmarking results: no obfuscation

BIBLIOGRAPHY

[1] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. *Deobfuscation: From Virtualized Code Back to the Original.* In Proceedings of the 15th Conference on Detection of Intrusions and Malware, and Vulnerability. 2018

[2] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, Alexander Pretschner *Code Obfuscation Against Symbolic Execution Attacks.* ACSAC '16 Proceedings of the 32nd Annual Conference on Computer Security Applications. Pages 189-200. 2016

[3] Cristina Cifuentes and K. John Gough. *Decompilation of binary programs.* Software - Practice & Experience, 25(7):811-829, July 1995.

[4] Christian Collberg, Clark Thomborson, and Douglas Low *A taxonomy of obfuscating transformations.* Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[5] Schwarts, E. J., Avgerinos, T., and Brumley, D. All *You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask).* In IEEE Symposium on Security and Privacy (2010).

[6] Godefroid, P., Klarlund, N., Sen, K. *DART: directed automated random testing.* In PLDI 2005. ACM

[7] *Vmprotect.* http://vmpsoft.com (2003–2017)

[8] *Codevirtualizer.* https://oreans.com/codevirtualizer.php

[9] *Tigress: C diversifier/obfuscator.* http://tigress.cs.arizona.edu/

[10] *Tigress_protection.* https://github.com/JonathanSalwan/Tigress_protection

[11] Xu, Hui and Zhou, Yangfan and Kang, Yu and Tu, Fengzhi and Lyu, Michael *Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution*

[12] *ImageZero* https://github.com/cfeck/imagezero