Grenoble INP — **UGA**
Master CybserSecurity — **Year 2023-24**

Advanced Security (L. Mounier)

Written Exam - Tuesday January the 16th, 2024

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Electronic devices are forbidden.

This exam contains two distinct parts: 2 exercises and some questions on a research paper.

Exercise 1. (∼ 5 pts)

We consider a Java class `ReadFile` which, when executed, needs to read files from the user[1] home directory. Figure 1 gives successively the source codes of `ReadFile.java` and the one of a calling class `C1.java`.

```
1  import java.lang.*;
2  import java.security.*;
3
4  class ReadFile {
5   public ReadFile () {
6    try {
7      String fileName= ...   // name of the file to open
8      FileReader fileReader = new FileReader(fileName); // open fileName
9    } catch (Exception e) {
10      System.err.println("Caught exception " + e.toString());
11   }
12  }
13 }
```

```
1  import java.lang.*;
2  import java.security.*;
3
4  class C1 {
5    public static void main(String[] args) {
6        ReadFile rf  = new ReadFile() ; // calls ReadFile
7    }
8  }
```

Figure 1: The source codes of ReadFile and of one of its calling class C1

Q1. We compile `ReadFile.java` and `C1.java` in a same directory `D1`. Then we run `C1` (still in this same directory). Everything works fine, namely the user files can be read normally.

The developer now wants to strictly restrict the use of `ReadFile` to calling classes compiled **within D1 only**, such classes being considered as trustable enough. To do so he introduces a dedicated *security policy* controlled by a *security manager*.

[1]the one running the application calling this class

1. What information do you need to provide inside the security policy description file?
2. As a developer, how would you test that this proposed solution ensures the expected security requirements?

You do not need to provide the exact Java syntax nor command lines, but your answer should be as precise as possible ...

```
import java.lang.*;
import java.security.*;

// Trusted class, compiled within D1
class C2 {
 public C2() // calls ReadFile
   {ReadFile rf = new ReadFile() ;}
}
```

```
import java.lang.*;
import java.security.*;

// Untrusted class
class C3 {
 public static void main(String[] args)
   {C2 c2 = new C2() ;} // calls C2
}
```

Figure 2: The source codes of callers `C2.java` and `C3.java`

Q2. We now consider a second scenario with a chain of two callers, where `C3` calls `C2` which calls `ReadFile`, as illustrated on Figure 2. We assume that `C2` is compiled within `D1`, but not `C3`, which is compiled and executed from elsewhere.

1. What will happen when running `C3`[2], knowing that `C2` is **allowed** to call `ReadFile`?
2. What is the strategy used by Java to produce this result (or, more generally, to decide whether the security policy is violated or not)?

Q3. Finally, the developer wants to allow **any** `C2` caller to call `ReadFile`, assuming that `C2` will perform enough checks, but he still wants to use a security manager and the same security policy to keep a strict control over other callers of `ReadFile`.

How could he proceed to do that, and what are the benefits and drawbacks of your proposed solution?

Q4. Another option would have been to use OS-level (e.g., Linux) permissions to control the access of the user home directory. In particular we could restrict the access of the user home directory only to applications those owner is allowed to do so (according to Linux permissions). What would be the advantages and drawbacks (if any!) of this approach with respect to the one described in the previous questions.

Exercise 2. (∼ 5 pts)

The piece of code given Figure 3 is part of a program supposed to be (routinely) **ran as root** on a machine.

```
...
access(filename)      // check filename access
fd = open(filename) // open filename in file descriptor fd
write(fd,...)       // write inside the file referred by fd
...
```

Figure 3: A vulnerable C code

[2] with the security manager and security policy of **Q1**

We give below an excerpt of the `access()` function man page:

`int access(const char *pathname, int mode);`
access() checks whether the calling process can access the file pathname.
If pathname is a symbolic link, it is dereferenced.
Return value:
On success (all requested permissions granted and the file exists), zero is returned.
Warning:
Using these calls to check if a user is authorized to, for example, open a file before actually doing so using open(2) creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. For this reason, ***the use of this system call should be avoided****.*

Q1. Explain the meaning of the last sentence of this man page, and why is it consider as a security hole (telling what gain an attacker could obtain on this particular example of Figure 3).

Q2. Give a concrete attack setup, providing:

- an informal description of a potential attacker objective and behavior (in a few lines);
- a C program example he/she could use to run this attack (if necessary);
- the (realistic) hypothesis you need to assume for this attack to succeed.

You can make use for instance of the two following C functions:

int link(const char *oldpath, const char *newpath)
link() creates a new link (also known as a hard link) to an existing file. If newpath exists, it will not be overwritten. This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

int unlink(const char *pathname)
unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse. If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed. If the name referred to a symbolic link, the link is removed.

Q3. To prevent the attack mentioned in the previous question, a possible solution is first to perform **multiple consecutive checks**, opening the file into **distinct** file descriptors after each check, and then verifying that all the opened files are (physically) identical before writing into it (through the opened file descriptor).

1. Explain why this solution may (or must ?) work.
2. Update the code of Figure 3 to implement it; yo can use the macro `CMP(fd1,fd2)` to verify that files referred by `fd1` and `fd2` are physically identical.

Questions on a research paper. (∼ 10 pts)

Read the paper given in appendix **up to section 3.6 (included)** answering the following questions as long as you read it. When answering these questions you should not copy entire sentences from the paper but rather illustrate your point with examples and comments from your own.

Q1 [section 1].

- Briefly summarize the main approaches mentionned in the paper to defeat virtualization-based obfuscation.
- Give (and briefly explain) some other obfuscation techniques than virtualization you are aware of.

Q2 [section 2.2]. Using Listing1.1 and 1.2, give the (virtual) bytecode that would be obtained from the following C statement, assuming that `x`, `y` and `z` are located respectively in registers `r0`, `r1`, `r2`:

```
x = y + z*2
```

Q3 [section 2.3]. Why is it said that *Recovering 100% of the original binary code is impossible in general*?

Q4 [section 3.1]. Can you explain the difference betwen **"direct"** tainting and **"indirect"** tainting (which is similar to "explicit" vs "implicit" data-flow).

Q5 [sections 3.2 and 3.3].
We replace the first statement of Listing 1.4 by `int var1 = x` (instead of `int var1 = 1`), still assuming that `x` is tainted. Give the new version of Figure 4.

Q6 [section 3.4 to 3.6].

1. Obviously, **full path coverage** cannot be achieved on an arbitrary code. Why?
2. In section 3.1 the authors said they consider **direct tainting** only. Do you think it could be an issue with respect to the correctness of P^*?

Q7 (more general question). What are the main limitations you can see regarding the proposed approach? And, as a defender, how could we proceed to prevent the use of this de-obfuscation technique (still using virtualization-based obfuscation)?

Symbolic deobfuscation: from virtualized code back to the original[⋆] (long version)

Jonathan Salwan[1], Sébastien Bardin[2], and Marie-Laure Potet[3]

[1] Quarkslab, Paris, France
[2] CEA, LIST, Univ. Paris-Saclay, France
[3] Univ. Grenoble Alpes, F-38000 Grenoble, France
`jsalwan@quarkslab.com, sebastien.bardin@cea.fr,`
`marie-laure.potet@univ-grenoble-alpes.fr`

Abstract. Software protection has taken an important place during the last decade in order to protect legit software against reverse engineering or tampering. *Virtualization* is considered as one of the very best defenses against such attacks. We present a generic approach based on symbolic path exploration, taint and recompilation allowing to recover, from a virtualized code, a devirtualized code semantically identical to the original one and close in size. We define criteria and metrics to evaluate the relevance of the deobfuscated results in terms of correctness and precision. Finally we propose an open-source setup allowing to evaluate the proposed approach against several forms of virtualization.

1 Introduction

Context. The field of software protection has increasingly gained in importance with the growing need of protecting sensitive software assets, either for pure security reasons (e.g., protecting security mechanisms) or for commercial reasons (e.g., protecting licence checks in video games or video on demand). Virtual machine (VM) based software protection (a.k.a. *virtualization*) is a modern technique aiming at transforming an original binary code into a custom Instruction Set Architecture (ISA), which is then emulated by a custom interpreter. Virtualization is considered as a very powerful defense against reverse engineering and tampering attacks, taking a central place during the last decade in the software protection arsenal [24, 3–5].

Attacking virtualization. In the same time, researchers have published several methods to analyze such protections. They can be partitioned into semi-manual approaches [14, 17, 21], automated approaches [12, 18, 23, 25, 26] and program synthesis [11, 29]. Semi-manual approaches consist in manually detecting and understanding VM's opcode handlers, and then, writing a dedicated disassembler. They rely on the knowledge of the reverse engineer and they are time

[⋆] Work partially funded by ANR and PIA under grant ANR-15-IDEX-02.

consuming. Some classes of automated approaches aim at automatically reconstructing the (non-virtualized) control flow of the original program, but they require to detect some virtualization artefacts [12, 23] (virtual program counter, dispatcher, etc.) – typically through some dedicated pattern matching. These approaches must be adapted when new forms of virtualization are met. Finally, another class of approaches [7, 25] tries to directly reconstruct the behaviors of the initial code (before virtualization), based on trace analysis geared at eliminating the virtualization machinery. Such approaches aim to be agnostic with respect to the different forms of virtualization. Yet, while the ultimate goal of deobfuscation is to recover the original program, these approaches focus rather on intermediate steps, such as identifying the Virtual Machine machinery or simplifying traces.

Goal & challenges. While most works on devirtualization target malware detection and control flow graph recovery, *we focus here on sensitive function protections (such as authentication), either for IP or integrity reasons*, and we consider the problem of fully recovering the original program behavior (expurged from the VM machinery) and compiling back a new (devirtualized) version of the original binary. We suppose we have access to the protected (virtualized) function and we are interested in recovering the original non-obfuscated code, or at least a program very close to it. We consider the following open questions:

– How can we characterize the relevance of the deobfuscated results?
– How much can such approaches be independent of the virtualization machinery and its protections?
– How can virtualization be hardened against such approaches?

Contribution. Our contributions are the following:

– We present a fully automatic and generic approach to devirtualization, based on combining taint, symbolic execution and code simplification. We clearly discuss limitations and guarantees of the proposed approach, and we demonstrate the potential of the method by automatically solving (the non-jitted part of) the Tigress Challenge in a completely automated manner.
– We design a strong experimental setup[1] for the systematic assessment of the qualities of our framework: well-defined questions & metrics, a delimited class of programs (hash-like functions, integrity checks) and adequat measurement besides code similarities (full correctness). We also propose a systematic coverage of classic protections and their combinations.
– Finally, we propose an open-source framework based on the Triton API, resulting in reproducible public results.

[1] Solving the Tigress Challenge was presented at the French industrial conference SSTIC'17 [19]. The work presented here adds a revisited description of the method, a strong systematic experimental evaluation as well as new metrics to evaluate the accuracy of the approach.

The main features of our approach are summarized in Figure 1, in comparison with others works. In particular we propose and discuss some notions of correctness and completeness as well as a set of metrics illustrating the accuracy of our approach. Fig. 1 will be explained in more details in Sec. 6.

	manual	Kinder[12]	Coogan[7]	Yadegari[26]	Our approach
identify input	required	required	required	required	required
understand vpc	required	required	no	no	no
understand dispatcher	required	no	no	no	no
understand bytecode	required	no	no	no	no
output	simplified CFG	CFG + invariants	simplified trace	simplified CFG	simplified code
key techno.	-	static analysis (abstract interp.)	value-based slicing	taint, symbolic code slicing instr. simplification	taint, symbolic formula slicing formula simplification code simplification
xp: type of code	-	toy examples	toys+malware	toys+malware	hash functions
xp: #samples	-	1	12	44	920
xp: evaluation metrics	-	known invariants	%simplification	similarity	size, correctness

Fig. 1: Position of our approach

Discussion. While our approach still shows limitations on the class of programs that can be handled (cf. Section 5), the present work clearly demonstrates that hash-like functions (typical of proprietary assets protected through obfuscation) can be easily retrieved from their virtualized versions, challenging the common knowledge that virtualization is the best defense against reversing – while it is true for a human attacker, it does not hold anymore for an automated attacker (unless the defender is ready to pay a high running time overhead with deep nested virtualization). Hence, defenders must take great care of protecting the VM machinery itself against semantic attacks.

Long version. This version adds a discussion on the (implicit) *backward slicing* step performed at formula level (Section 3.3), and it also presents more detailed statistics about the Tigress challenge (Tables 6 and 7 in Section 4.5), in order to ease reproducibility and comparison of results.

2 Background: Virtualization and Reverse Engineering

2.1 Virtualization-based Software Protection

Virtualization-based software protections aim at encoding the original program into a new binary code written in a custom Instruction Set Architecture (ISA) shipped together with a custom Virtual Machine (VM). Such protections are offered by several industrial and academic tools [24, 3–5]. Generally, it is composed of 5 principal components, close to CPU design (Figure 2):

1. **Fetch**: Its role is to fetch, from the VM's internal memory, the *(virtual) opcode* to emulate, based on the value of a *virtual program counter* (`vpc`).
2. **Decode**: Its role is to decode the fetched opcode and its appropriate operands to determine which ISA instruction will be executed.

3. **Dispatch**: Once the instruction is decoded, the dispatcher determines which *handler* must be executed and sets up its context.
4. **Handlers**: They emulate virtual instructions by sequences of native instructions and update the internal context of VM, typically `vpc`.
5. **Terminator**: The terminator determines if the emulation is finished or not. If not, the whole process is executed one more time.

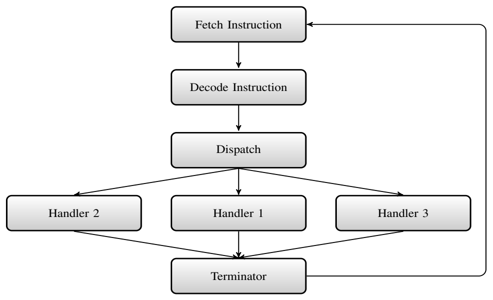

Fig. 2: Standard Virtual Machine Architecture

2.2 Example

Let us consider the C function of Listing 1.1 we want to virtualize it. Disassembly of the VM's bytecode is in comment in Listing 1.1. Once Listing 1.1 is compiled to VM's bytecode, it must be interpreted by the virtual machine itself. The sample of code illustrated by Listing 1.2 could be this kind of VM. The VM is called with an initial `vpc` pointing to the first opcode of the bytecode (e.g: the virtual address of instruction `mov r0, r9`). Once the opcode has been fetched and decoded by the VM, the dispatcher points to the appropriate handler to virtually execute the instruction and then, the handler increments `vpc` to point on the next instruction to execute and so on until the virtualized program terminate. As we can see, the control flow of the original program is lost and replaced by a dispatcher pointing on all VM's handlers (here, only four instructions).

2.3 Manual De-virtualization

Manual devirtualization typically comes down to writing a disassembler for the (unknown) virtual architecture under analysis. It consists of the following steps:

1. Identify that the obfuscated program is virtualized, and identify its input;
2. Identify each component of the virtual machine;
3. Understand how all these components are related to each other, especially which handler corresponds to which bytecode, the associated semantics, where operands are located and how they are specified;
4. Understand how `vpc` is orchestrated.

```
int func(int x) {
  int a = x;
  int b = 2;
  int c = a * b;
  return c;
}


/*
** Bytecodes equivalence:
**
** 31 ff 00 09:  mov r0, r9
** 31 01 02 00:  mov r1, 2
** 44 00 00 01:  mul r0, r0, r1
** 60:           ret
*/
```

Listing 1.1: A C function

```
void vm(ulong vpc, struct vmgpr* gpr) {
  while (1) {
    /* Fetch and Decode */
    struct opcode* i = decode(fetch(vpc));
    /* Dispatch */
    switch (i->getType()) {
      /* Handlers */
      case ADD /* 0x21 */:
        gpr->r[i->dst] = i->op1 + i->op2;
        vpc += 4; break;
      case MOV /* 0x31 */:
        gpr->r[i->dst] = i->op1;
        vpc += 4; break;
      case MUL /* 0x44 */:
        gpr->r[i->dst] = i->op1 * i->op2;
        vpc += 4; break;
      case RET /* 0x60 */:
        vpc += 1; return;
}}}
```

Listing 1.2: Example of VM

Once all these points have been addressed, we can easily create a specific disassembler targeted to the virtual architecture. Yet, solving each step is time consuming and may be heavily influenced by the reverse engineer expertise, the design of the virtual machine (e.g: which kind of dispatcher, of operands, etc.) and the level of obfuscation implemented to hide the virtual machine itself.

Discussion. Recovering 100% of the original binary code is impossible in general, that is why devirtualization aims at proposing a binary code as close as possible to the original one. Here, we seek to provide a semantically equivalent code expurged from the components of the virtual machine (*devirtualized code*). In other words, starting from the code in Listing 1.2, we want to derive a code semantically equivalent and close (in size) to the code in Listing 1.1.

3 Our Approach

We rely on the key intuition that *an obfuscated trace T' (from the obfuscated code P') combines original instructions from the original code P (the trace T corresponding to T' in the original code) and instructions of the virtual machine VM such that $T' \triangleq T + VM(T)$*. If we are able to distinguish between these two subsequences of instructions T and $VM(T)$, we then are able to reconstruct one path of the original program P from a trace T'. By repeating this operation to cover all paths of the virtualized program, we will be able to reconstruct the original program P – in case the original code has a finite number of executable paths, which is the case in many practical situations involving IP protection.

3.1 Overview

The main steps of our approach, sketched in Fig. 3, are the following ones:

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.

In our approach, Step-0 (identifying input) must still be done manually, in a traditional way. By input we include all kinds of external interactions depending on the user, such as environment variables, program arguments and system calls (e.g. `read`*,* `recv`*, etc.). Analysts will typically rely on tools such as IDA or debuggers for this step.*

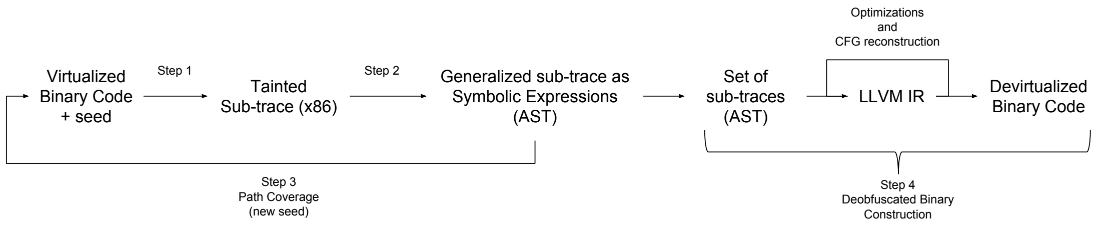
Virtualized Binary Code + seed
Step 1
Tainted Sub-trace (x86)
Step 2
Generalized sub-trace as Symbolic Expressions (AST)
Set of sub-traces (AST)
Optimizations and CFG reconstruction
LLVM IR
Devirtualized Binary Code
Step 3 Path Coverage (new seed)
Step 4 Deobfuscated Binary Construction

Fig. 3: Schematized Approach

Our approach is based on the tool suite Triton [20] which provides several advanced classes to improve dynamic binary analysis, in particular a concolic execution engine, a SMT symbolic representation and a taint analysis engine.

Dynamic Symbolic Execution. (DSE) [22, 9, 10] (a.k.a. concolic execution) is a technique that interprets program variables as symbolic variables along an execution. During a program execution, the DSE engine builds arithmetic expressions representing data operations and logical expressions characterizing path constraints along the execution path. These constraints can then be solved automatically by a constraint solver [27] (typically, SMT solver) in order to obtain new input data covering new paths of the program. Conversely to pure symbolic execution, DSE can reduce the complexity of these expressions by using concrete values from the program execution (*"concretization"* [10]).

Dynamic Taint Analysis. (DTA) [6, 28] aims to detect which data and instructions along an execution depend on user input. We consider direct tainting. Regarding the code in Listing 1.3 where user input is denoted by `input`, we start by tainting the input at line 1. Then, according to the instruction semantics, the taint is spread into `rax` at line 1, then `rcx` at line 3 and `rdi` at line 4. To resume, using a taint analysis, we know that instructions at line 1, 3, and 4 are in interaction with user input, while other lines are not.

```
1. mov rax, input
2. mov rcx, 1
3. add rcx, rax
4. mov rdi, rcx
```

Listing 1.3: x86 ASM sample

Taint can be combined with symbolic execution in order to explore all paths depending on inputs, resulting in input values covering these paths.

3.2 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 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.

3.3 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.

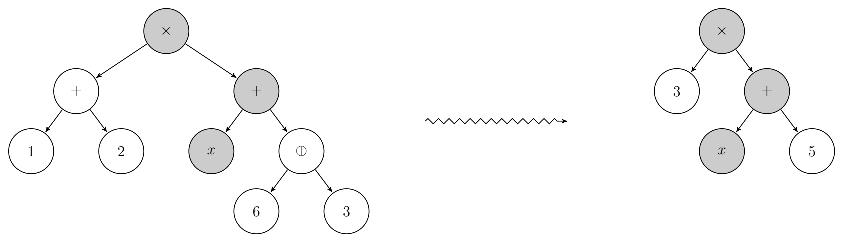

Fig. 4: Concretization of non tainted expressions

In order to better understand what Step 2 does, let us consider the function illustrated in Listing 1.4. Variable `x` is tainted as well as symbolized and the expression associated to variable `var8` is illustrated on the left of Figure 4 (gray nodes are tainted data). Then, once we concretize all un-tainted nodes, the expression becomes the one illustrated on the right. This mechanism typically allows to remove the VM machinery.

```
int f(int x) {
    int var1 = 1;
    int var2 = 2;
    int var3 = var1 + var2;
    int var4 = 6;
    int var5 = 3;
    int var6 = var4 ^ var5;
    int var7 = x + var6;
    int var8 = var3 * var7;
    return var8;
}
```

Listing 1.4: Sample of C code

A note on formula-level backward slicing. *As it is common in symbolic execution*, the symbolic representation is first computed in a forward manner along the path (see [15, Figure 2] for the basic algorithm), then all logical operations and definitions affecting neither the *final result* nor the followed path are removed from the symbolic expression (formula slicing, a.k.a. formula pruning – see for example [16]). This turns out to perform on the formula the equivalent of a backward slicing code analysis from the program output.

3.4 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 [10] 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 5). 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.

3.5 Step 4 - Generate a New Binary Version

At this step we have all 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 (AST in Fig. 3) and compile

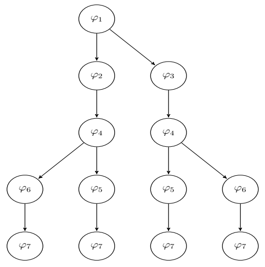

Fig. 5: Path Tree

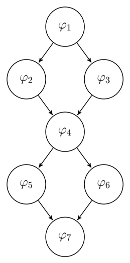

Fig. 6: A Reconstructed CFG

it. In particular we benefit from all LLVM (code level) optimizations[2] to partially rebuild a simplified Control Flow Graph (Figure 6). 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 to devirtualize it to an ARM architecture.

3.6 Guarantees: About Correctness and Completeness

Let P be the obfuscated program and $P^\star$ the extracted program. We want to guarantee that P and $P^\star$ behave equivalently for each input. We decompose this property into two sub-properties:

- local correctness: for a given input i, P and $P^\star$ behave equivalently,
- completeness: local correctness is established for each input.

While local correctness can often be guaranteed, depending on properties of each step (see Figure 7), completeness is lost in general as it requires full path exploration of the virtualized program. Interestingly enough, it can be recovered in the case of programs with a small number of paths, which is the case for many typical hash or crypto functions.

Step	Component	flaw	threats on $P^\star$
1	taint	undertainting overtainting	incorrect too large
2	path predicate	under-approximated over-approximated	incomplete incorrect
3	path exploration	incomplete	incomplete
4	code optimization	incorrect incomplete	incorrect too large

Fig. 7: Impact of each components on the overall approach

[2] Such as `simplifycfg` and `instcombine`.

3.7 Implementation

We develop a script[3] implementing our method. The Triton library [20] is in charge of everything related to the DSE and the taint engine. We also use Arybo [8] to move from the Triton representation to the LLVM-IR [13] and the LLVM front-end to compile the new binary code. The Triton DSE engine is standard [10, 22]: paths are explored in a depth-first search manner, memory accesses are concretized *à la* DART [10] (resulting in incorrect concretization) [15], logical formulas are expressed in the theory of bitvectors and sent to the Z3 SMT solver. Triton is engineered with care and is able to handle execution traces counting several dozen millions of instructions.

Regarding the discussion in Section 3.6, we can state that our implementation is correct on programs without any user-dependent memory access, and that it is even complete if those programs have a small number of paths (say, less than 100). While very restrictive, these conditions do hold for many typical hash-like functions, representative of proprietary assets protected through obfuscation.

4 Experiments

In order to evaluate our approach we proceed in two steps. First we carry out a set of systematic controlled experiments in order to precisely evaluate the key properties of our method (Sections 4.1 to 4.4). Second we address a real life deobfuscation challenge (Tigress Challenge) in order to check whether our approach can address uncontrolled obfuscated programs (Section 4.5). Code, benchmarks and more detailed results are available online[4]. We propose the three following evaluation criteria for our deobfuscation technique:

C_1: **Precision**,
C_2: **Efficiency**,
C_3: **Robustness w.r.t. the protection**.

4.1 Controlled Experiment: Setup

Our test bench is composed of 20 hash algorithms comprising 10 well-known hash functionsand 10 homemade ones taken from the Tigress Challenge[5] (see Table 1). The proposed functions are typically composed of a statically-bounded loop and contains one or two execution paths. These programs are typical of the kinds of assets the defender might want to protect in a code.

In order to protect these 20 samples, we choose the open-use binary protector Tigress[6], a diversifying virtualizer/obfuscator for the C language that supports many novel defenses against both static and dynamic reverse engineering and

[3] https://github.com/JonathanSalwan/Tigress_protection/blob/master/solve-vm.py
[4] https://github.com/JonathanSalwan/Tigress_protection
[5] Thanks to Christian Collberg for having provided us the original source codes.
[6] http://tigress.cs.arizona.edu