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.

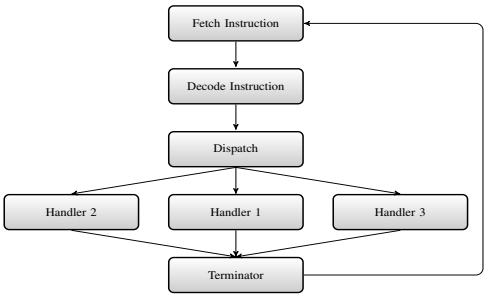

Fig. 2: Standard Virtual Machine Architecture

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.*

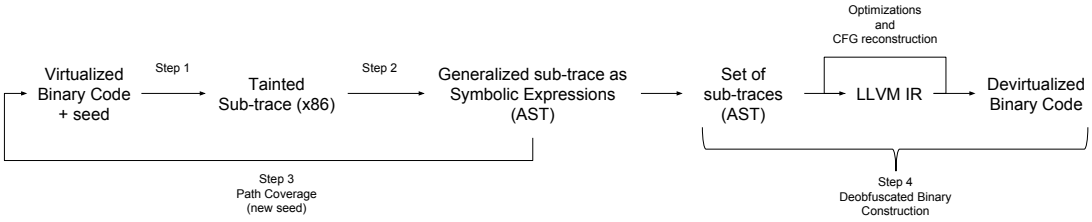

Fig. 3: Schematized Approach

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.

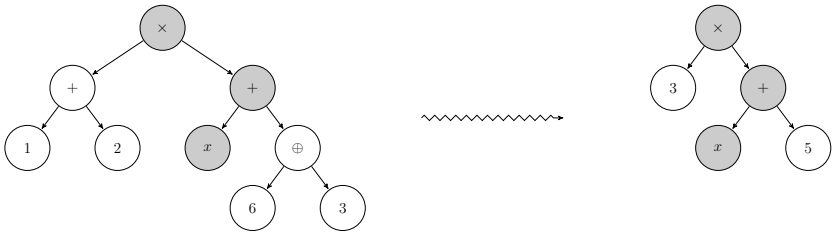

Fig. 4: Concretization of non tainted expressions

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

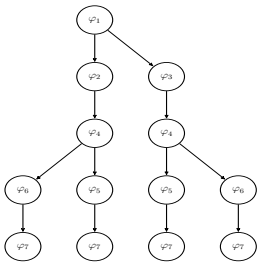

Fig. 5: Path Tree

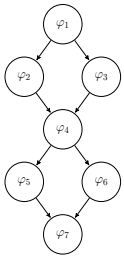

Fig. 6: A Reconstructed CFG

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

Hash	Loops	Binary Size (inst)	# executable paths
Adler-32	✓	78	1
CityHash	✓	175	1
Collberg-0001-0	✓	167	1
Collberg-0001-1	×	177	2
Collberg-0001-2	×	223	1
Collberg-0001-3	✓	195	1
Collberg-0001-4	✓	183	1
Collberg-0004-0	×	210	2
Collberg-0004-1	×	143	1
Collberg-0004-2	✓	219	2
Collberg-0004-3	✓	171	1
Collberg-0004-4	✓	274	1
Fowler-Noll-Vo Hash (FNV1a)	×	110	1
Jenkins	✓	79	1
JodyHash	✓	90	1
MD5	✓	314	1
SpiHash	✓	362	1
SpookyHash	✓	426	1
SuperFastHash	✓	144	1
Xxhash	✓	182	1

Table 1: List of virtualized hash functions for our benchmark

devirtualization attacks. Then, we select all virtualization-related binary protections (46) and apply each of them on each of the 20 samples, yielding a total benchmark of 920 protected codes (see Table 2). The goal is then to retrieve an equivalent and devirtualized version of each protected code. All these tests are applied on a Dell XPS 13 laptop with a Intel i7-6560U CPU, 16GB of RAM and 8GB of SWAP on a SSD.

Protecticons	Options
Anti Branch Analysis	goto2push, goto2call, branchFuns
Max Merge Length	0, 10, 20, 30
Bogus Function	0, 1, 2, 3
Kind of Operands	stack, registers
Opaque to VPC	true, false
Bogus Loop Iterations	0, 1, 2, 3
Super Operator Ratio	0, 0.2, 0.4, 0.6, 0.8, 1.0
Random Opcodes	true, false
Duplicate Opcodes	0, 1, 2, 3
Dispatcher	binary, direct, call, interpolation, indirect, switch, ifnest, linear
Encode Byte Array	true, false
Obfuscate Decode Byte Array	true, false
Nested VMs	1, 2, 3

Table 2: Tigress Protections

4.2 Precision (C_1)

The C_1 criterion aims to determine two points 1. **correctness:** is the deobfuscated code semantically equivalent to the original code? 2. **conciseness:** is the size of the deobfuscated code similar to the size of the original code?

Metrics used: Regarding correctness, after applying our approach we test over 4,000 integer inputs (the 1000 smallest integers, the 1000 largest ones, 2000 random others) whether the two corresponding output (obfuscated and deobfuscated) are identical or not. If yes, we consider the deobfuscated code as semantically equivalent. We also manually check 50 samples taken at random. Regarding conciseness, we consider the number of instructions before and after protections, and then after devirtualization.

Results: Table 3 gives an average of ratios (in term of number of instructions) between the original code and the obfuscated one, and also between the original code and the deobfuscated one. This table demonstrates that 1. after applying our approach, we are able to reconstruct valid binaries (in term of correctness) for 100% of our samples; 2. after applying protections, the sizes of binaries and traces are considerably increased and after applying our approach we reconstruct binaries sometimes slightly smaller than the original ones. This phenomenon is due to the fact that we concretize everything not related to the user input (Step 2), including initialisation and set up. Manual inspections also reveal that when the original code does not contain any loop, the recovered code exhibits almost the same CFG as the original code.

	Original	Obfuscated	Deobfuscated
Binary Size	min: 78 max: 426 avg: 196	min: 468 max: 5,424 avg: 1,205	min: 48 max: 557 avg: 119
Trace Size	min: 92 max: 9,743 avg: 726	min: 1,349 max: 47,927,795 avg: 229,168	min: 48 max: 557 avg: 143

(a) Sizes

	Original → Obfuscated	Original → Deobfuscated
Correctness		100%
Binary Size	min: x3.3 max: x14.0 avg: x6	min: x0.1 max: x2.8 avg: x0.71
Trace Size	min: x17 max: x1252 avg: x424	min: x0.05 max: x0.9 avg: x0.39

(b) Size ratios

Table 3: Size and correctness (920 samples)

Conclusion: Our approach does allow to recover semantically-equivalent devirtualized codes in all cases, with sizes very close to those of the original codes (even slightly smaller in average, despite loop unrolling), thus drastically decreasing the size of the protected code. Interestingly, our devirtualized codes have also simpler execution traces than the original codes.

4.3 Effectiveness (C_2)

The C_2 criterion aims at determining the effectiveness of our approach in terms of absolute time (required amount of resources) and also in trend (scalability).

Metrics used: We took measure at each step of our analysis and at each 10,000 instructions handled. These metric results can be found in detail in Table 9 (Appendix) and its *Obfuscated* (trace size) and *Time* columns.

Results: Figure 8 is the time-step of our approach on the 920 samples. About 80% of samples take less than 5 seconds to be deobfuscated. The most difficult example takes about 1h10 for approximatively 48 millions of instructions (MD5 with two levels of virtualization).

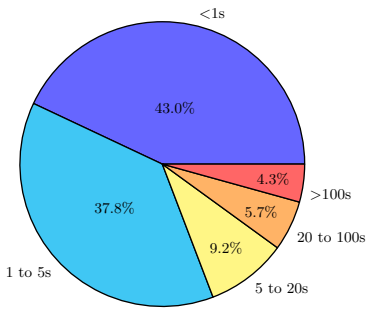

Fig. 8: Time-step (920 samples)

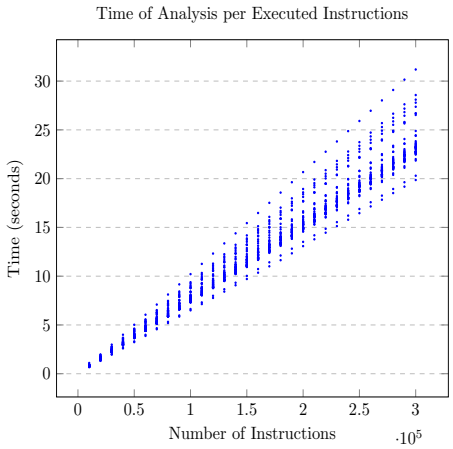

Fig. 9: Time w.r.t. number of instr. (all protections, MD5 algo.)

According to these results, we can see that the time taken by our analysis is linear w.r.t. the number of instructions on the obfuscated traces (*size of the execution tree*). If we focus on the MD5 example[7] and draw a dot at each 10,000 instructions handled and then for each protections, we get as a result Figure 9. Each dotted curve of this figure is one of the 46 protections used for our benchmark and each dot is a measure at each 10,000 instructions step. We can clearly see that curves possess a linear aspect.

Conclusion: Our approach has a linear time of analysis according to the number of explored instructions (execution tree), meaning that our approach does not add complexity w.r.t. standard DSE exploration. The more the protection integrates instructions in the binary the more our analysis will take time and RAM consuming but only with a constant evolution. Regarding our samples, we managed to devirtualize lot of them very quickly (only few seconds), and even for the hardest examples we were able to solve them in a short time on common hardware.

[7] MD5 is one of the most involving examples in our benchmark.

4.4 Influence of Protections (C_3)

This criterion aims at identifying whether certain specific protections do impact the analysis more than other protections (correctness, conciseness or performances), and if yes, how much.

Metrics Used: We consider the conciseness metrics, i.e. the number of instructions during the executions of the obfuscated binaries, the deobfuscated binaries and the original ones. We use them on the 46 different protections applied on the same hash algorithm, and then for all hash algorithms.

Results: According to Table 9 in appendix (*Deobfuscated* column), we can clearly answer that the conciseness is the same whatever protection is applied. We get the same result for each one of these protected binary codes. Protections do not influence the number of instructions recovered for all the 20 hash algorithms tested. As an example, Figure 10 illustrates the influence of different dispatchers analyzed on the MD5 example and we can clearly see that the number of instructions recovered is identical whatever dispatcher is applied. Moreover, previous results in Section 4.3 (Figure 9) have already demonstrated that all considered protections have an effect on efficiency directly proportional to the increase they involve on the trace size.

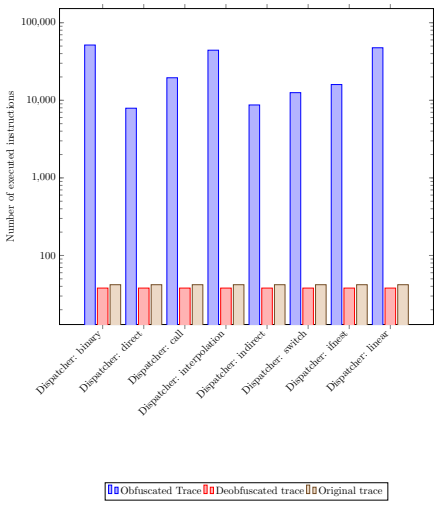

Fig. 10: Influence of dispatchers on our analysis

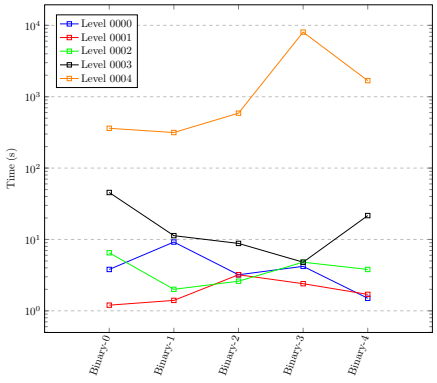

Fig. 11: Time to solve each Tigress challenge

Conclusion: Our approach, in term of precision, is not influenced by the chosen protections and our outputs are identical whatever the protections applied. Yet, as already shown, the protection can influence the analysis time and make the analysis intractable. The previous section shows that such a protection can be effective only if it implies a large runtime overhead - which can be a severe problem on some applications. For example regarding the MD5 example, the execution overhead is 10x with 1 level of VM, 100x with 2 and 6800x with 3.

Discussion on each protection In order to really understand why our approach works on such protections, we open a discussion for each category of them.

Complicated VM machinery (opaque vpc, dispatchers, etc.): These protections are mainly introduced to slowdown a static analysis. Yet, using a dynamic taint analysis (Step 1 of Section 3.2), we are able to distinguish which instructions are dedicated to the virtual machine and which instructions emulate the original behavior of the program (*pertinent instructions*). The virtual machine's subexpressions are then eliminated through concretization in Step 2 (see Section 3.3).

Duplicate Opcodes: This protection makes the VM more complicated to understand by a human, but it does not prevent its exploration (Steps 1 to 3). In our experiments, duplicated opcodes are identified and merged together because of code-level (compiler) optimizations (Step 4) together with the normalization induced by the transformation to symbolic expressions (Step 2).

Nested VM: As already discussed, nesting VMs does not impact the precision but the performance of our method. Hence the defender can indeed prevent the attack, but it comes at a high cost as the running time overhead for the defender is directly proportional to the analysis time overhead for the attacker. As an example, we are able to solve up to 2 nested levels with our setup machine (16GB of RAM), but we solve 3 nested levels using an Amazon EC2 instance.

4.5 Case Study: The Tigress Challenge

We have chosen the Tigress Challenge as a case study to demonstrate that our approach works even in presence of strong combinations of protections. The challenge[8] consists of 35 virtual machines with different levels of obfuscation (Table 4). All challenges are identical: there is a virtualized hash function $f(x) \rightsquigarrow x'$ where x is an integer and the goal is to recover, as close as possible, the original hash algorithm (all algorithms are custom). According to their challenge status, only challenge `0000` had been previously solved and October 28th, 2016 we published [9] a solution for challenges `0000` to `0004` with a presentation at SSTIC

[8] http://tigress.cs.arizona.edu/challenges.html#current
[9] http://tigress.cs.arizona.edu/index.htm

2017 [19] (each challenge contains 5 binaries, resulting in 25 virtual machine codes). We do not analyze jitted binaries (`0005` and `0006`) as jit is not currently supported by our implementation.

Challenge	Description	Difficulty	Web Status	Our Status
0000	One level of virtualization, random dispatch.	1	Solved	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	2	Open	**Solved**
0002	One level of virtualization, bogus functions, implicit flow.	3	Open	**Solved**
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	2	Open	**Solved**
0004	Two levels of virtualization, implicit flow.	4	Open	**Solved**
0005	One level of virtualization, one level of jitting, implicit flow.	4	Open	Open[†]
0006	Two levels of jitting, implicit flow.	4	Open	Open[†]

[†]: Jit not supported by our script.

Table 4: Tigress Challenge (each challenge contains 5 virtual machines)

We have been able to automatically solve all the aforementioned open challenges in a correct, precise and efficient way, demonstrating that the good results observed in our controlled experiments extend to the uncontrolled case. Correction has been checked with random testing and manual inspection. Figure 11 illustrates the time and memory consumption for each challenge – again time and memory consumption are proportional to the number of instruction executed. The hardest challenge family is `0004` with two levels of virtualization. For instance, challenge `0004-3` contains 140 millions of instructions, reduced to 320 in 2 hours (see Tables 5, 6 and 7). Additional details can be found in [19].

	Tigress challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	3.85s	9.20s	3.27s	4.26s	1.58s
0001	1.26s	1.42s	3.27s	2.49s	1.74s
0002	6.58s	2.02s	2.63s	4.85s	3.82s
0003	45.6s	11.3s	8.84s	4.84s	21.6s
0004	361s	315s	588s	8049s	1680s

Table 5: Time (in seconds) to solve Tigress Challenge

5 Discussion

We first summarize limitations of our approach, together with possible mitigations. Then we discuss how our technique can be defended against.

	Tigress challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	x0.85	x1.09	x0.73	x0.89	x1.4
0001	x0.41	x0.60	x0.26	x0.22	x0.53
0002	x0.29	x0.28	x0.51	x1.40	x0.42
0003	x1.10	x1.17	x1.57	x0.46	x0.44
0004	x0.81	x0.38	x0.70	x0.37	x0.53

Table 6: Ratio (size) original → deobfuscated

	Tigress challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	675 → 242	754 → 309	838 → 275	668 → 245	792 → 343
0001	11690 → 257	11864 → 283	16506 → 266	13721 → 247	11585 → 284
0002	1356 → 257	1742 → 260	1717 → 274	1399 → 310	1345 → 279
0003	10135 → 534	9996 → 354	17812 → 396	6039 → 276	10504 → 312
0004	10776 → 337	5980 → 249	6304 → 327	6773 → 258	11480 → 338

Table 7: From obfuscated to deobfuscated in term of number of instructons

5.1 Limits and mitigations

The main limitation of our method is that it is mostly geared at programs with a small number of paths. In case of a too high number of paths, large parts of the original code may be lost, yielding an incomplete recovery. Yet, we are considering here *executable paths* rather than syntactic paths in the CFG, and we already made the case that hash and other cryptographic functions often have only very few paths – only one path in the case of timing-attack resistant implementations.

Also our current implementation is limited to programs without any user-dependent memory access. This limitation can be partly removed by using a more symbolic handling of memory accesses in DSE [15], yet the tainting process will have to be updated too. Since we absolutely want to avoid undertainting (see Figure 7, Section 3.6), dynamic tainting will have to be complemented with some form of range information. Note that we require only direct tainting, limiting the undertainting effect.

Another class of limitations arises from programs using features beyond the scope of our symbolic reasoning, such as multithreading, intensive floating-point arithmetic reasoning, self-modification, system calls, etc. Extending to these constructs is hard in general as it may require significant advances in symbolic reasoning. Note, however, that there are some recent progress in floating-point arithmetic reasoning, and that (simple) self-modification can be handled quite directly in DSE [1]. Moreover, regarding system calls, adequate modelling of the environment could be useful here – not that much a research question, but a clearly manpower-intensive task. Finally, while completeness is clearly out of

scope here, local correctness can still be enforced in many cases by relying on the concretization part of DSE.

Note also that while bounded loops and non-recursive function calls are handled, they are currently recovered as inlined or unrolled code, yielding a potential blowup of the size of the devirtualized code. It would be interesting to have a postprocessing step trying to rebuild these high-level abstractions.

5.2 Potential defenses

Protecting the VM by attacking our steps. As usual, deobfuscation approaches may be broken by attacking their weaknesses. It is actually a never-ending cat-and-mouse game. Figure 7 (Section 3.6) gives a good idea of the kind of attacks our method can suffer from. As the first step of our approach reposes on a taint analysis aiming at isolating pertinent instructions, a simple defense could be to spread the taint into VM's components like `decoder` or `dispatcher`. The more the taint is interlaced with VM components, the less our approach will be precise, as tainted data are symbolized. Especially if we symbolize `vpc` our path exploration step will run into the well-known path explosion problem. We can also imagine a defense based on hash functions over jump conditions (e.g: `if (hash(x) == 0x1234)`) which will break constraint solvers during path exploration. Precise dynamic tainting and more robust crypto-oriented solvers are current hot research topics. Another possibility is to implement anti-dynamic tricks to prevent tracing. This issue is more an engineering problem, but it is not that easy to handle well.

In a general setting, symbolic attacks and defenses are a hot topic of deobfuscation, and several protections against symbolic reasoning have been investigated. Any progress in this domain can be directly re-used, either for or against our method. Yet, these protections are not that easy to implement well, and it is sometimes hard to predict whether they will work fine or not. Especially, the protections have to depend on user input, otherwise they will be discarded by taint analysis. Note also that we do not claim that our method can overcome all of these defenses: we focus only on the virtualization step.

Protecting the bytecode instead of the VM. Another interesting defense is to protect the bytecode of the virtual machine instead of its components. Thus, if the virtual machine is broken, the attacker gets as a result an obfuscated pseudo code. For example, this bytecode could be turned into unreadable Mixed Boolean Arithmetic (MBA) expressions.

6 Related work

Several heuristic approaches to devirtualization have been proposed (e.g., [23]), yet our work is closer to semantic devirtualization methods [7, 26, 12]. It has also connexions with recent works on symbolic deobfuscation [2, 1, 8]. Figure 1 Section 1 gives a synthetic comparison of these different approaches.

Manual and heuristic devirtualization. Sharif *et al.* [23] propose a dynamic analysis approach which tries to identify `vpc` based on memory access patterns, then they reconstruct a CFG from this sequence of `vpc`. However, their method suffers from limitations. For example, their loop detection strategies are not directly applicable to emulators using a threaded approach. Their approach is also likewise not applicable to dynamic translation-based emulation. Another point is that their approach expects each unique address in memory to hold only one abstract variable, which means that an adversary may utilize the same location for different variables at different times to introduce imprecision in their analysis. Conversely, our method solves this problem since we are working on a trace over a SSA representation, making aliasing trivial to catch up. They also mention nesting virtualization as an open problem, while our method has been shown to handle some level of nesting.

Semantics devirtualization. Coogan *et al.* [7] focus on identifying instructions affecting the observable behavior of the obfuscated code. They propose a dynamic approach based on a form of tainting together with leveraging the knowledge from system calls and ABIs. In the end, they identify a subtrace of the virtualized trace containing only those instructions affecting the program output. Their approach can devirtualize only a single path (the executed one) and cannot be applied on virtualized functions without any system call.

Yadegari *et al.* [26] proposes a generic approach to deobfuscation combining tainting, symbolic execution and simplifications. Their goal is to recover the CFG of obfuscated malware, and they carry out experimental evaluation with several obfuscation tools. Our technique shows similarities with their own approach, yet we consider the problem of recovering back a semantically-correct (unprotected) binary code in typical cases of IP protections (hash functions), and we perform a large set of controlled experiments, regarding all virtualization options provided by the Tigress tool, in order to evaluate the properties of our approach.

Kinder [12] proposes a static analysis based on abstract interpretation built over a `vpc`-sensitive abstract domain. Its approach performs a range analysis on the whole VM interpreter, providing the reverser with invariants on the arguments of function calls.

Symbolic deobfuscation. Banescu *et al.* [2] recently evaluate the efficiency of standard obfuscation mechanisms against symbolic deobfuscation. They conclude, as we do, that without any proper anti-symbolic trick these defenses are not efficient. They also propose a powerful anti-symbolic defense mechanism, but it requires some form of secret sharing and thus falls outside the strict scope of man-at-the-end scenario we consider here. These two works are complementary in the sense that we focus only on virtualization-based protection, but we cover it in a more intensive way and we take a more ambitious notion of deobfuscation (get back an equivalent and small code) while they consider program coverage. In the same vein, recent promising results have been obtain by symbolic deobfuscation against several classes of protections [1, 8, 26].

7 Conclusion and Future Work

We propose a new automated dynamic analysis geared at fully recovering the original program behavior of a virtualized code – expurged from the VM machinery, and compiling back a new (devirtualized) version of the original binary. We demonstrate the potential of the method on small hash-like functions (typical of proprietary assets protected by obfuscation) through an extensive experimental evaluation, assessing its precision, efficiency and genericity, and we solve (the non-jitted part of) the Tigress Challenge in a completely automated manner. While our approach still shows limitations on the class of programs that can be handled, this work clearly demonstrates that hash-like functions can be easily retrieved from their virtualized versions, challenging the common knowledge that virtualization is the best defense against reversing.

In a near future we will focus on the reconstruction of more complicated program structures such as user-dependent loops or memory accesses.

References

1. Bardin, S., David, R., Marion, J.-Y.: Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. S&P 2017. IEEE
2. Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., Pretschner, A.: Code obfuscation against symbolic execution attacks. In ACSAC 2016.
3. Codevirtualizer. https://oreans.com/codevirtualizer.php
4. Themida. https://www.oreans.com/themida.php
5. Tigress: C diversifier/obfuscator. http://tigress.cs.arizona.edu/
6. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA 2007. ACM
7. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: CCS 2011. ACM
8. Eyrolles, N., Guinet, A., Videau, M.: Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. In: GreHack 2016.
9. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating software testing using program analysis. IEEE Software 25(5), 30–37 (2008)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In PLDI 2005. ACM
11. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE 2010. ACM/IEEE
12. Kinder, J.: Towards static analysis of virtualization-obfuscated binaries. In: 19th Working Conference on Reverse Engineering, WCRE 2012
13. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. 2004
14. Maximus: Reversing a simple virtual machine. CodeBreakers 1.2 (2006)
15. David, R., Bardin, S., Feist, J., Mounier, L., Potet, M-L., Thanh Dinh Ta, Marion, J-Y.: Specification of concretization and symbolization policies in symbolic execution. In ISSTA 2016. ACM
16. David, R., Bardin, S., Ta, T. D., , Feist, J., Mounier, L., Potet, M.-L., Marion, J.-Y.: BINSEC/SE : A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. In SANER 2016. IEEE

17. Rolles, R.: Defeating hyperunpackme2 with an ida processor module (2007)
18. Rolles, R.: Unpacking virtualization obfuscators. In: WOOT 2009
19. Salwan, J., Bardin, S., Potet, M.L.: Deobfuscation of vm based software protection. In: SSTIC 2017
20. Saudel, F., Salwan, J.: Triton: A dynamic symbolic execution framework. In: SSTIC 2015
21. Scherzo: Inside code virtualizer (2007)
22. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: FSE 2005
23. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Automatic reverse engineering of malware emulators. In: S&P 2009. IEEE
24. Vmprotect: http://vmpsoft.com (2003–2017)
25. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: CCS 2015.
26. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: S&P 2015. IEEE
27. Vanegue, J., Heelan, S., Rolles, R.: SMT Solvers in Software Security. WOOT 2012.
28. Schwartz, E. J., Avgerinos, T., Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: S&P 2010. IEEE
29. Blazytko, t., Contag, M., Aschermann, C., Holz, T.: Syntia: Synthesizing the Semantics of Obfuscated Code. In:USENIX Security Symposium 2017. Usenix

A Detailed experiments

	Traces Size (instructions)			Binary Size (instructions)		
Protection	**Original**	**Obfuscated**	**Deobfuscated**	**Original**	**Obfuscated**	**Deobfuscated**
Anti Branch Analysis: branchFuns	min: 92 max: 9,743 avg: 698	min: 3,047 max: 1,555,703 avg: 121,460	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 935 max: 2,048 avg: 1,641	min: 48 max: 599 avg: 122
Kind of Operands: stack	min: 92 max: 9,743 avg: 698	min: 1,430 max: 381,230 avg: 31,104	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 783 max: 1,139 avg: 979	min: 48 max: 599 avg: 122
Kind of Operands: registers	min: 92 max: 9,743 avg: 698	min: 1,459 max: 425,285 avg: 34,322	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 807 max: 1,182 avg: 1,065	min: 48 max: 599 avg: 122
Opaque to VPC: False	min: 92 max: 9,743 avg: 698	min: 1,430 max: 381,230 avg: 31,104	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 783 max: 1,139 avg: 979	min: 48 max: 599 avg: 122
Opaque to VPC: True	min: 92 max: 9,743 avg: 698	min: 1,600 max: 700,138 avg: 51,405	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 861 max: 1,296 avg: 1,148	min: 48 max: 599 avg: 122
Duplicate Opcodes: 3	min: 92 max: 9,743 avg: 698	min: 1,430 max: 381,230 avg: 31,037	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 783 max: 1,226 avg: 1,063	min: 48 max: 599 avg: 122
Dispatcher: binary	min: 92 max: 9,743 avg: 698	min: 2,449 max: 2,825,359 avg: 195,969	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 814 max: 1,154 avg: 1,010	min: 48 max: 599 avg: 122
Dispatcher: interpolation	min: 92 max: 9,743 avg: 698	min: 2,625 max: 2,592,186 avg: 181,698	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 839 max: 1,183 avg: 1,037	min: 48 max: 599 avg: 122
Dispatcher: linear	min: 92 max: 9,743 avg: 698	min: 2,115 max: 5,804,970 avg: 351,747	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 785 max: 1,125 avg: 982	min: 48 max: 599 avg: 122
Nested VMs: 1	min: 92 max: 9,743 avg: 698	min: 1,430 max: 381,230 avg: 30,104	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 783 max: 1,139 avg: 979	min: 48 max: 599 avg: 122
Nested VMs: 2	min: 92 max: 9,743 avg: 698	min: 37,479 max: 47,927,795 avg: 3,520,624	min: 48 max: 599 avg: 122	min: 78 max: 426 avg: 192	min: 676 max: 1,182 avg: 814	min: 48 max: 599 avg: 122

Table 8: Average of all algorithms per protection

	Traces Size (instructions)			Binary Size (instructions)					
Hash	**Original**	**Obfuscated**	**Deobfuscated**	**Original**	**Obfuscated**	**Deobfuscated**	**Time (s)**	**RAM (KB)**	**Correctness**
Adler-32	min: 235 max: 235 avg: 235	min: 5,385 max: 2,996,678 avg: 169,174	min: 222 max: 222 avg: 222	min: 78 max: 78 avg: 78	min: 665 max: 2,001 avg: 1,092	min: 222 max: 222 avg: 222	min: 0.4 max: 516.0 avg: 26.6	min: 84,784 max: 2,469,276 avg: 203,737	100%
CityHash	min: 200 max: 200 avg: 200	min: 1,455 max: 37,532 avg: 3,555	min: 57 max: 57 avg: 57	min: 175 max: 175 avg: 175	min: 571 max: 1,396 avg: 938	min: 57 max: 57 avg: 57	min: 0.1 max: 3.2 avg: 0.3	min: 81,664 max: 93,540 avg: 82,756	100%
Collberg-0001-0	min: 173 max: 173 avg: 173	min: 4,497 max: 2,840,513 avg: 174,703	min: 79 max: 79 avg: 79	min: 167 max: 167 avg: 167	min: 679 max: 3,366 avg: 1,243	min: 79 max: 79 avg: 79	min: 0.4 max: 494.7 avg: 26.4	min: 86,008 max: 2,339,380 avg: 204,447	100%
Collberg-0001-1	min: 326 max: 326 avg: 326	min: 8,456 max: 2,066,306 avg: 103,599	min: 167 max: 167 avg: 167	min: 177 max: 177 avg: 177	min: 685 max: 3,697 avg: 1,300	min: 96 max: 96 avg: 96	min: 0.8 max: 184.2 avg: 9.3	min: 102,948 max: 883,780 avg: 136,964	100%
Collberg-0001-2	min: 227 max: 227 avg: 227	min: 7,099 max: 2,132,169 avg: 104,401	min: 84 max: 84 avg: 84	min: 223 max: 223 avg: 223	min: 685 max: 5,043 avg: 1,364	min: 84 max: 84 avg: 84	min: 0.6 max: 182.0 avg: 9.3	min: 93,016 max: 899,528 avg: 127,183	100%
Collberg-0001-3	min: 262 max: 262 avg: 262	min: 7,467 max: 2,071,933 avg: 98,637	min: 68 max: 68 avg: 68	min: 195 max: 195 avg: 195	min: 687 max: 4,367 avg: 1,342	min: 68 max: 68 avg: 68	min: 0.6 max: 164.8 avg: 8.0	min: 90,400 max: 898,128 avg: 122,732	100%
Collberg-0001-4	min: 228 max: 228 avg: 228	min: 5,881 max: 1,510,030 avg: 107,831	min: 100 max: 100 avg: 100	min: 183 max: 183 avg: 183	min: 709 max: 3,676 avg: 1,315	min: 100 max: 100 avg: 100	min: 0.5 max: 168.1 avg: 12.5	min: 86,564 max: 896,148 avg: 145,051	100%
Collberg-0004-0	min: 372 max: 372 avg: 372	min: 10,348 max: 7,804,232 avg: 465,758	min: 190 max: 190 avg: 190	min: 210 max: 210 avg: 210	min: 702 max: 3,631 avg: 1,317	min: 99 max: 99 avg: 99	min: 1.1 max: 1431.0 avg: 74.4	min: 116,452 max: 6,306,932 avg: 435,567	100%
Collberg-0004-1	min: 147 max: 147 avg: 147	min: 3,810 max: 859,278 avg: 46,031	min: 67 max: 67 avg: 67	min: 143 max: 143 avg: 143	min: 636 max: 2,704 avg: 1,165	min: 67 max: 67 avg: 67	min: 0.3 max: 71.0 avg: 4.1	min: 85,904 max: 420,912 avg: 100,100	100%
Collberg-0004-2	min: 408 max: 408 avg: 408	min: 11,294 max: 2,999,784 avg: 138,738	min: 332 max: 332 avg: 332	min: 219 max: 219 avg: 219	min: 722 max: 4,765 avg: 1,400	min: 128 max: 128 avg: 128	min: 1.6 max: 275.2 avg: 13.2	min: 172,760 max: 1,243,348 avg: 206,449	100%
Collberg-0004-3	min: 203 max: 203 avg: 203	min: 5,503 max: 1,439,344 avg: 96,331	min: 78 max: 78 avg: 78	min: 171 max: 171 avg: 171	min: 718 max: 3,478 avg: 1,317	min: 78 max: 78 avg: 78	min: 0.5 max: 138.9 avg: 11.1	min: 86,948 max: 755,056 avg: 137,375	100%
Collberg-0004-4	min: 307 max: 307 avg: 307	min: 8,674 max: 9,279,883 avg: 533,675	min: 146 max: 146 avg: 146	min: 274 max: 274 avg: 274	min: 725 max: 5,424 avg: 1,452	min: 146 max: 146 avg: 146	min: 0.7 max: 1,681.6 avg: 86.6	min: 103,964 max: 7,480,952 avg: 482,005	100%
FNV1a	min: 143 max: 143 avg: 143	min: 1,499 max: 54,846 avg: 3,544	min: 57 max: 57 avg: 57	min: 110 max: 110 avg: 110	min: 517 max: 1,180 avg: 861	min: 57 max: 57 avg: 57	min: 0.1 max: 4.9 avg: 0.3	min: 80,872 max: 101,828 avg: 82,139	100%
Jenkins	min: 201 max: 201 avg: 201	min: 5,520 max: 1,069,111 avg: 76,420	min: 125 max: 125 avg: 125	min: 79 max: 79 avg: 79	min: 631 max: 1,888 avg: 1,076	min: 125 max: 125 avg: 125	min: 0.5 max: 83.9 avg: 6.2	min: 87,572 max: 543,272 avg: 110,694	100%
JodyHash	min: 92 max: 92 avg: 92	min: 1,349 max: 155,637 avg: 9,820	min: 48 max: 48 avg: 48	min: 90 max: 90 avg: 90	min: 468 max: 1,085 avg: 803	min: 48 max: 48 avg: 48	min: 0.1 max: 25.2 avg: 1.4	min: 79,732 max: 203,072 avg: 86,237	100%
MD5	min: 9,743 max: 9,743 avg: 9,743	min: 173,673 max: 47,927,795 avg: 2,328,114	min: 557 max: 557 avg: 557	min: 314 max: 314 avg: 314	min: 1,311 max: 4,828 avg: 1,857	min: 557 max: 557 avg: 557	min: 16.5 max: 4,226.7 avg: 207.5	min: 266,032 max: 2,688,976 avg: 583,198	100%
SpiHash	min: 364 max: 364 avg: 364	min: 2,880 max: 1,694,015 avg: 100,661	min: 160 max: 160 avg: 160	min: 362 max: 362 avg: 362	min: 824 max: 1,829 avg: 1,224	min: 160 max: 160 avg: 160	min: 0.3 max: 288.1 avg: 15.1	min: 89,356 max: 1,434,764 avg: 159,257	100%
SpookyHash	min: 536 max: 536 avg: 536	min: 1,784 max: 140,565 avg: 9,571	min: 79 max: 79 avg: 79	min: 426 max: 426 avg: 426	min: 788 max: 1,443 avg: 1,125	min: 79 max: 79 avg: 79	min: 0.1 max: 23.1 avg: 1.3	min: 82,424 max: 193,080 avg: 88,364	100%
SuperFastHash	min: 182 max: 182 avg: 182	min: 1,402 max: 37,479 avg: 3,502	min: 81 max: 81 avg: 81	min: 144 max: 144 avg: 144	min: 506 max: 1,331 avg: 874	min: 81 max: 81 avg: 81	min: 0.1 max: 3.1 avg: 0.3	min: 82,572 max: 94,696 avg: 83,540	100%
Xxhash	min: 186 max: 186 avg: 186	min: 1,672 max: 103,193 avg: 9,310	min: 68 max: 68 avg: 68	min: 182 max: 182 avg: 182	min: 691 max: 1,470 avg: 1,047	min: 68 max: 68 avg: 68	min: 0.1 max: 16.3 avg: 1.1	min: 83,376 max: 164,128 avg: 88,478	100%

Table 9: Average of all protections per hash function