Intertwining ROP Gadgets and Opaque Predicates for Robust Obfuscation

Fukutomo Nakanishi,*   Giulio De Pasquale,   Daniele Ferla,†   Lorenzo Cavallaro

King's College London   **King's College London and Toshiba Corporation*   *†Università di Bologna*

Abstract

Software obfuscation plays a crucial role in protecting intellectual property in software from reverse engineering attempts. While some obfuscation techniques originate from the obfuscation-reverse engineering arms race, others stem from different research areas, such as binary software exploitation.

Return-oriented programming (ROP) gained popularity as one of the most effective exploitation techniques for memory error vulnerabilities. ROP interferes with our natural perception of a process control flow, which naturally inspires us to repurpose ROP as a robust and effective form of software obfuscation. Although previous work already explores ROP's effectiveness as an obfuscation technique, evolving reverse engineering research raises the need for principled reasoning to understand the strengths and limitations of ROP-based mechanisms against man-at-the-end (MATE) attacks.

To this end, we propose ROPFuscator, a fine-grained obfuscation framework for C/C++ programs using ROP. We incorporate opaque predicates and constants and a novel instruction hiding technique to withstand sophisticated MATE attacks. More importantly, we introduce a realistic and unified threat model to thoroughly evaluate ROPFuscator and provide principled reasoning on ROP-based obfuscation techniques that answer to code coverage, incurred overhead, correctness, robustness, and practicality challenges.

1 Introduction

Software has been transforming the fabric of our society for decades. It is now virtually impossible to imagine any activity that does not involve software components to some extent. As such, software is widely recognized as an important intellectual property to protect from reverse engineering attempts [1–3].

In this context, obfuscation represents the de-facto standard when it comes to protecting software from being disclosed. However, although useful, there is no clear winner in the obfuscation-reverse engineering arms race [4]. Thus, relying

on an arsenal of obfuscation techniques seems to be the only effective way to counteract attempts to break such confidentiality requirements. Dummy code insertion [3], control flow flattening [2], self-modifying code [5], opaque predicates [6], virtualization [7], and anti-debugging [8] are well-known software obfuscation techniques. They all exploit assumptions that challenge—one way or another—the logic of reverse engineering algorithms. For instance, the insertion of dummy code and opaque predicates interferes with the attempt to directly reconstruct the semantics of an underlying algorithm. In contrast, control flow flattening breaks the ability to understand a program's execution flow, which challenges further reasoning to identify properties of interest.

While advances in reverse engineering spin the creation of more sophisticated obfuscation techniques, others result from intriguing leaps from different research areas. In this context, return-oriented programming (ROP) gained popularity as one of the most advanced memory error exploitation technique [9]. Core to this is the ability to chain the invocation of chunks of code (gadgets) to execute arbitrary (often malicious) code. As such, ROP builds its working logic on threaded code [10], changing the usual interpretation we have of code execution centered on the instruction pointer, for one pivoted on the stack pointer.

This observation naturally suggests ROP can be repurposed to represent a robust and effective form of software obfuscation. First, threaded code changes our understanding of control flow graphs, de-facto breaking subsequent data-flow analysis that relies on them. Second, ROP provides fine control on the granularity of obfuscation, being able to operate at the level of individual assembly instructions. Third, an obfuscated piece of code would see its semantics as the result of the execution of code gadgets scattered potentially throughout the entire process address space.

Prior work already explores the effectiveness of ROP as an obfuscation technique. For instance, Mohan et al. [11] and Borrello et al. [12] repurpose ROP to challenge malware detection tasks. Conversely, Mu et al. [13] apply ROP to obfuscate a program's control flow graph at a coarse granularity,

ignoring fine-grained obfuscation of individual assembly instructions. Despite being promising, they all fail at exploring the assumptions and the extents to which ROP represents a viable solution for obfuscation techniques to withstand man-at-the-end (MATE) reverse engineering attacks [14], where reverse engineering attempts tailored at ROP-based obfuscation [15, 16] or dynamic symbolic execution [17] still undermine its effectiveness.

These challenges highlight the need for principled reasoning to understand the strengths and limitations of ROP-based obfuscation mechanisms against MATE attacks. We propose ROPFuscator, a framework to obfuscate C/C++ programs. At its core, ROPFuscator relies on ROP micro-gadgets [18] to obfuscate arbitrary C/C++ programs at the granularity of individual assembly instructions. To withstand MATE attacks of increasing sophistication, ROPFuscator relies on opaque predicate and constants [19] and a novel approach to intertwine ROP gadgets of arbitrary length with such opaque constructs, thus challenging the ability to distinguish between the two and thus reconstruct the original program's semantics. We present a thorough evaluation of ROPFuscator across 5 dimensions, which support our principle reasoning with evidence on completeness (code coverage), incurred overhead, correctness, robustness to MATE attacks, and practicality. We release ROPFuscator to further support the need for principled reasoning in domains characterized by endemic attack-defense arms-race.

In summary, we make the following contributions:

- We introduce a unified threat model that ROP-based obfuscation techniques must address to assess their robustness to increasingly sophisticated MATE attacks (§2.1). This helps us to provide a principle reasoning to identify and justify the design choices that avoid brittle arms-race that are anyway endemic to the software obfuscation domain, and instead of providing researchers and practitioners with a clear and contextualized understanding of strengths and limitations (§2.2.1-2.2.3).

- We present ROPFuscator, a framework for fine-grained obfuscation of C/C++ programs with ROP (§3.2). To withstand sophisticated MATE attacks, we equip ROP-Fuscator with opaque predicates and constants (§3.3), and we build a novel instruction hiding technique that intertwines ROP gadget of arbitrary length in opaque predicates to challenge analysis in distinguishing between the two and thus the semantics of the obfuscated code against code to withstand analyses (§3.4).

- We present a thorough evaluation of ROPFuscator along 5 dimensions to support our principled reasoning and provide the opportunity to understand its effectiveness in practical contexts (§4).

2 ROPFuscator

We aim at providing principled reasoning on the use of ROP to obfuscate real-world programs. Particular care is placed on the description of a realistic threat model, which drives the design choices of ROPFuscator and identifies the research questions one must answer to understand the assumptions and extents to which we can consider ROP as a realistic technique for programs obfuscation.

2.1 Threat Model

Our threat model considers MATE attacks of increasing sophistication [14]. In particular, we assume attackers can rely on static ROP-agnostic and ROP chain disassembly analyses as well as dynamic symbolic execution and ROP-specific dynamic analyses. In doing so, we adopt metrics similar to the one used in [3, 20].

For simplicity and ease referencing these threats to motivate the design choices and support the underlying principled reasoning and evaluation, we refer to the following threats as **Threat A-D**. However, they should not be seen as individual and disconnected threat models. On the contrary, they represent a realistic and unified threat model that explores how robust ROPFuscator is in facing adaptive attacks that are aware of ROPFuscator inner working mechanisms.

Threat A: ROP-agnostic Static Analysis. The core to static analysis of binary programs is disassembly. Linear sweep and recursive traversal are two main static disassembly algorithms that aim to recover a program's assembly instructions by analyzing a sequence of bytes linearly (linear sweep) or following the expected execution flow (recursive traversal). Decompilation is often built on a successful disassembly to convert assembly code into high-level program constructs.

Threat B: Static ROP Chain Analysis. It is perhaps unsurprising that ROP chains' introduction in a program naturally breaks traditional disassembly algorithms. In fact, they assume an IP-centric code execution model, whereas ROP, built on threaded code, focuses on an SP-centric one. A more realistic threat model here should thus consider the ability of statically analyzing ROP chains to reconstruct the obfuscated program original control flow one can leverage to build more insightful data-flow and decompilation analyses on.

Threat C: Dynamic Symbolic Execution. Static ROP chain analysis requires to identify the address of ROP gadgets. Address-agnostic ROP gadgets, therefore, challenge this analysis effectively. The mechanism to build ROP chains to hide ROP gadgets is not straightforward but is discussed thoroughly in the next sections. Here, it is enough to assume this is a possibility. Therefore, our threat model must include attacks that rely on dynamic symbolic execution (DSE) to identify such information. Once successful, one can rely on the above analyses to recover the original program's semantics.

Threat D: Dynamic ROP Chain Analysis. This analysis takes advantage of runtime information in a context in which the attacker knows ROP is a core building block for program obfuscation. Instruction traces collected from a running process are passed to a CPU emulator, which executes the ROP chains, extracting the original code from the gadgets [16, 21].

2.2 Design Choices

The following section provides insights on the design of the code transformation we realize in the context of program obfuscation. We first rely on the insertion of ROP chains to obfuscate the code of interest. In particular, we rely on the use of ROP micro-gadgets [18] and target obfuscation at a fine granularity, from individual instructions to basic blocks and entire functions of interest. This code transformation addresses Threat A. To withstand Threat B, we rely on the use of opaque predicates and constants [19]. This helps in concealing ROP gadgets' address, challenging any attempt to reconstruct ROP logics with static analyses. Similarly, we address Threat C by choosing specific opaque predicates and constants as thoroughly outlined in §3.3. Finally, we intertwine ROP chains and opaque predicates code to withstand sophisticated attacks that rely on dynamic ROP chain analysis (Threat D).

2.2.1 ROP Transformation

The first component's purpose is to convert an arbitrary portion of code into ROP chains. The resulting code has a significantly different structure and can already disrupt the automated analysis performed by decompilers.

The gadgets are extracted from libraries linked by the compiler (e.g., libc) or provided by the user. Later, the instructions are matched with a semantically equivalent set of previously identified gadgets to finalize the translation. The transformation is also applied to branch instructions, allowing the obfuscation of code *inside* and *between* basic blocks.

Further details are described in §3.2.

2.2.2 Opaque Predicate Insertion

One of the challenges in adopting ROP as an obfuscation technique is its relative fragility against static analysis. Once a ROP chain is identified, it is possible to reconstruct the original code and defeat the obfuscation [15]. For this reason, we interpose opaque predicates and opaque constants in the ROP chain generation to improve its robustness against static analysis.

The culprit of statically analyzing a ROP chain is to find its gadget addresses. Therefore, we use opaque constants to compute and protect the addresses of said gadgets. Besides, we obfuscate the immediate operands of instructions when needed.

Original code	Hidden code inserted	Dummy code inserted
mov edx, 0xda598211	mov edx, 0xda598211	mov edx, 0xda598211
mul edx	mul edx	mul edx
(Insertion Point)	**mov ecx, 123**	**add [esp], 456**
cmp eax, 0x40527619	cmp eax, 0x40527619	cmp eax, 0x40527619
setne al	setne al	setne al
cmp edx, 0xde447238	cmp edx, 0xde447238	cmp edx, 0xde447238
setne dl	setne dl	setne dl

Figure 1: Assembly code of opaque predicates before/after instructions are inserted

Finally, opaque predicates are effective in hindering DSE tools [22, 23]. However, the design of the predicates has a significant impact on their robustness against symbolic analysis. In light of this, we evaluated the possible algorithms that best suit our needs, and we discuss their implementation in §3.3.

2.2.3 Instruction Hiding

The use of opaque predicates and opaque constants alone is not enough to protect our approach against dynamic analyses. For example, it is possible to isolate and extract the ROP chain by tracing the process's execution.

To avoid full disclosure of the code using instruction tracing, we hide part of the instructions in the code, which calculates the opaque predicates' output. The predicate's computation does not interfere with the context needed by the instruction being hidden, making the opaque predicate a code cave. Due to the predicate calculation's code size, it is possible to hide a few instructions without raising suspicion.

We demonstrate two examples of this technique in Figure 1. Each opaque predicate has several *insertion points*, which are used to store instructions that are not related to the calculation of the predicate. In these specific examples, we consider the situation where the user wants to obfuscate the instruction sequence mov ecx, 123; add edx, ecx while hiding mov ecx, 123 in opaque predicates. The instruction is inserted in the allotted insertion point, and a dummy instruction (add [esp], 456) is placed in a similar predicate to avoid identification by pattern matching. Using this approach, the ROP chain will not contain the entirety of the code, thwarting the analysis if the chain is extracted and analyzed.

Furthermore, this approach has better resilience to dynamic tracing compared to x86 variable-length instruction steganography. In the first case, the instructions of opaque predicates, gadgets, and dummy code are intertwined, and it becomes challenging to distinguish which one is which. On the other hand, once executed, instruction steganography directly reveals the hidden instructions, whereas it is only challenging in identifying them statically.
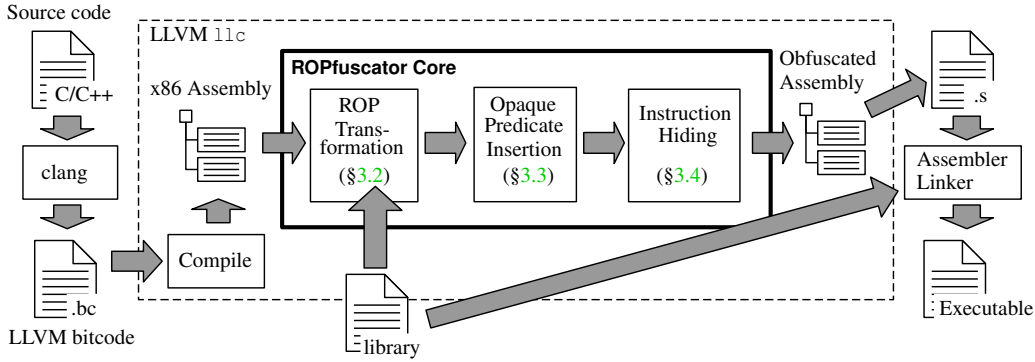
Figure 2: An architectural view of ROPFuscator

3 System Architecture and Implementation

We present a more detailed view of ROPFuscator to help understand its obfuscation steps and their interconnection. We also provide information on implementation pitfalls to help practitioners and developers working with binary programs.

3.1 Architectural Overview

Our framework obfuscates C/C++ code in x86 assembly level using LLVM. The source code is compiled to LLVM's IR and then processed by our framework. It consists of three components called in subsequent order, shown in Figure. 2, named ROP transformation, opaque predicate insertion and instruction hiding.

ROP transformation (§3.2) converts instructions into ROP chains. *Opaque predicate insertion* (§3.3) injects opaque predicates in the ROP chain generation code to protect the gadgets' entrypoint address. Finally, *instruction hiding* (§3.4) picks some instructions and embed them into opaque predicates.

The obfuscation components can be applied selectively while respecting their invocation order. For example, ROP transformation can be applied independently, while the opaque predicates pass cannot be used without first executing the ROP transformation.

3.2 ROP Transformation

Methods of converting normal code to equivalent gadgets are proposed in several studies [11, 24]. However, instead of processing native machine instructions, they are transforming various intermediate representations to ROP gadgets. In our work, we lift native x86 instructions. An obfuscation transform example is shown in Figure. 3.

The steps are explained in the following paragraphs.

Gadgets extraction The extraction process is based on the Galileo algorithm [9], and the gadgets are extracted from a shared library chosen by the user. For design simplicity,

we only rely on *microgadgets* [18] of length 1 (i.e only one instruction before the ret instruction) to build ROP chains.

ROP chain generation The use of microgadgets may incur in the unavailability of gadgets needed to perform operations on registers. For this reason, we decompose the original instruction in smaller computations that use temporary registers (Step (i) in Figure 3). The temporary registers are found by performing live register analysis [25] for each instruction within the basic blocks. Once the available registers are enumerated, we use gadgets to exchange them accordingly, similarly to the method proposed by Homescu et al. [18], to generate ROP chains (Step (ii) in Figure 3).

Emitting ROP Generator Code Once the gadgets are extracted, the ROP chain needs to be built and injected into the program. This is done by adding *rop generator code* which pushes the generated ROP chain onto the stack in reverse order, followed by a ret instruction (Step (iii) in Figure 3).

To deal with ASLR, a gadget address is calculated by using the address of a random symbol from the linked library as a base address. Later, the offset of the gadget address is added to the base address and then pushed to the stack. We do not use symbols defined in other linked libraries or the program to avoid symbol conflicts while computing the gadget addresses.

Instructions support It is important to translate as many instructions as possible into ROP chains since this directly affects the obfuscation scheme's robustness.

To find out which instructions are used the most, we observed the instruction count of various applications and used it as main metric. The most common instructions include non-control instructions (mov, add, xor, cmp, lea) and control instructions (jmp, call, je and jne). We implemented the support of all the most commonly used instructions in addition to others in order to achieve an high instruction coverage as shown in §4.1.
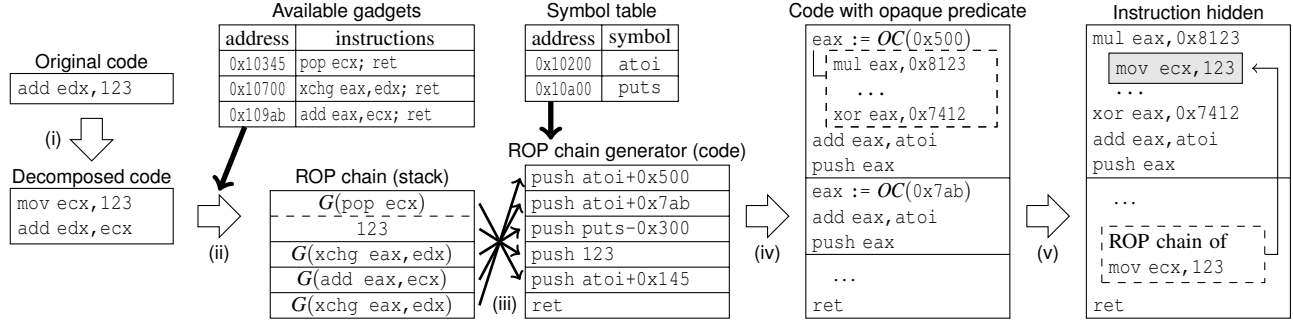
4

Original code

```
add edx,123
```

(i)

Decomposed code

```
mov ecx,123
add edx,ecx
```

(ii)

Available gadgets

address	instructions
0x10345	pop ecx; ret
0x10700	xchg eax,edx; ret
0x109ab	add eax,ecx; ret

ROP chain (stack)

```
G(pop ecx)
123
G(xchg eax,edx)
G(add eax,ecx)
G(xchg eax,edx)
```

(iii)

Symbol table

address	symbol
0x10200	atoi
0x10a00	puts

ROP chain generator (code)

```
push atoi+0x500
push atoi+0x7ab
push puts-0x300
push 123
push atoi+0x145
ret
```

(iv)

Code with opaque predicate

```
eax := OC(0x500)
  mul eax,0x8123
  ...
  xor eax,0x7412
add eax,atoi
push eax
eax := OC(0x7ab)
add eax,atoi
push eax
...
ret
```

(v)

Instruction hidden

```
mul eax,0x8123
mov ecx,123
...
xor eax,0x7412
add eax,atoi
push eax
...
ROP chain of
mov ecx,123
ret
```

Figure 3: Obfuscation Transformation Example with ROP, Opaque Predicates and Instruction Hiding

Program Semantics Preservation It is crucial to preserve the semantics of the program while obfuscating. The x86 ISA contains many instructions that modify the flag registers. Therefore, we use the pushf and popf instructions to save and restore the flags on necessity. Additionally, LLVM exposes useful metadata to check whether register flags may be safely clobbered. This provided us the choice to omit flag saving to improve performance.

3.3 Opaque Predicate Insertion

This section will discuss how opaque predicates are generated and later inserted in the ROP generation code. Besides, we detail how static and dynamic analysis affects opaque predicates and our approach to improving their resilience.

Several opaque predicates generation algorithms have been proposed in previous works. They are based on arithmetic operations, non-determinism [6], one-way functions [26] and computationally hard programs (e.g. pointer-aliasing [6], 3SAT [19, 27]).

We tested the use of integer factorization and the 3SAT problem for generating opaque predicates.

The algorithm based on *integer factorization* takes two 32-bit inputs x, y and returns 0 if $xy = C$ for a fixed 64-bit prime integer C, returning 1 otherwise. The factorization of C is needed to force that the output is always 1. This task is considered difficult if C is very large and it is used as a basis of the RSA cryptosystem [28].

The second algorithm, *Random 3-SAT*, is based on Sheridan et al. [27]. We generate a random conjunctive normal form (CNF) formula which consists of $32N$ clauses. The value of the factor N is taken according to the aforementioned results, which suggest $N \geq 6$ in order to have high chances for the clauses to be unsatisfiable. The formula is then negated, forcing the output to be always 1.

Based on internal results, opaque predicates that used random 3-SAT yield a worse size-to-performance ratio and were less robust against DSE. For these reasons, we excluded this algorithm from our final evaluation.

3.3.1 Opaque Gadget Addresses Against Static Analysis

We use opaque predicates to protect gadget addresses and immediate operands (Step (iv) in Figure 3). An opaque predicate can generate a 1-bit output which is hard to be computed statically.

For this reason, we concatenate each output bit of 32 distinct opaque predicate instances to generate a 32-bit constant (*opaque constant*). In our work, we focused on the x86 architecture, which uses 32-bit registers hence the choice to use 32 opaque predicates. Naturally, this approach can be easily extended to different architecture sizes. In this way, static analysis attacks, whether automated or manual, need to reverse engineer the appropriate number of opaque predicates to compute the protected value.

3.3.2 DSE-resistant Opaque Predicates

In the previous section, we discussed the generation of the opaque predicates and their application to protect the gadget addresses against static analysis. However, this is not robust enough against DSE attacks (threat C). In this section, we focus on the steps we undertook to make the predicates DSE-resistant. We evaluated our approach with angr [17], but we believe it can be extended to other DSE engines.

Concolic execution engines can execute code concretely; therefore, if the input to opaque predicates is statically known, the execution is deterministic. In this case, the engine can compute the opaque predicates' output very efficiently and, as a result, the gadget addresses. The ROP chain would then be exposed and executed as if it were not obfuscated in the first place. However, if the input is not known and concretized, the symbolic execution engine needs to symbolically evaluate the opaque predicates. This will force the computation of the mathematically hard problem on which the opaque predicate is based on.

Thus, we focused on finding appropriate input that cannot be easily concretized, imposing additional calculations to the symbolic execution engine.

Table 1: Example of contextual and invariant opaque predicates with integer factorization based algorithm

No.	type	predicate	input	output
1	Invariant	$xy = 531299$	Random	0
2	Invariant	$xy \neq 531299$	Random	1
3	Invariant	$xy = 531299$	Constant (any)	0
4	Invariant	$xy \neq 531299$	Constant (any)	1
5	Contextual	$xy = 428711$	Constant (577, 743)	1
6	Contextual	$xy = 428711$	Constant (otherwise)	0
7	Contextual	$xy \neq 428711$	Constant (577, 743)	0
8	Contextual	$xy \neq 428711$	Constant (otherwise)	1

Feeding Symbolic Input to Opaque Predicates DSE treats several kinds of values as symbolic, such as user input, generated random numbers, or the current timestamp. We opted to use user input since both random numbers and the current timestamp can be concretized under certain DSE exploration strategies. This approach is similar to the range divider mechanism used to diverge the control flow [22]. This method operates on user input, and it has been shown to effective against symbolic analysis.

However, it is challenging to define user input without having a higher-level view of the program semantics. For this reason, we compute input values by performing arithmetic operations on several general-purpose registers. We aim to hinder valid input inferring by DSE, as shown in the threat model in §2.1. Therefore, we assume that user input is used in the obfuscated code and that parts are processed into general-purpose registers, rendering the input symbolic. If our assumption is correct, this approach will increase the running time to an unusable extent.

We confirmed that feeding symbolic input increases DSE analysis time and memory usage. We evaluate this technique further in §4.3.

Using Invariant and Contextual Opaque Predicates There are several methodologies proposed to reverse engineer opaque predicates in order to replace them with 0 or 1 accordingly [29, 30].

These techniques have been previously classified into four categories [26]: brute-forcing or mathematical proof, direct substitution with 0 or 1, probabilistic substitution, and pattern matching. Building robust opaque predicates is out of the scope of this work; therefore, we do not define precise attack models in §2.1. However, we considered these attacks in our design, and we applied existing techniques to hinder them.

Brute-forcing or mathematical proofs can be impeded by choosing mathematically difficult problems, as explained above. To deal with both *direct and probabilistic substitutions*, we introduced contextual opaque predicates which change their output based on preconditions chosen at design-time [31]. Finally, we use crafted and random constants as input to the contextual opaque predicates to generate 0 or 1, respectively.

3.4 Instruction Hiding

Instruction hiding consists of three steps. After an instruction is decomposed into smaller operations, a subset of these instructions is marked to be hidden. Subsequently, the marked code is inserted into opaque predicates along with additional dummy code. We describe the process in more detail as follows.

Hidden Code Selection In this step, we decide which part of the code is to be obfuscated by instruction hiding or by ROP transformation. It is important to balance the two parts as an attacker can recover more instructions with execution tracing if ROP transformation is prevalent. On the other hand, the number of opaque predicates containing hidden instruction would be lower in the opposite case, leaving such instructions unprotected. We set a limit on the number of hidden instructions to be at most half of the total.

Embedding Code into Opaque Predicates The code is embedded into the opaque predicates through several insertion points. The insertion points are designed to minimize register and flag conflicts with the inserted code. However, if there is a clash, we use temporary registers to preserve semantics, as explained in §3.2.

Dummy Code Insertion As a final step, we inject dummy code to the remaining insertion points to diversify the opaque predicates, avoiding trivial pattern-matching detection. Additionally, the dummy code is intertwined with code crucial to the computation of the opaque predicate. The code other than computing useless operation modifies the predicate's internal state along with its variables, leading to added confusion for the attackers.

4 Evaluation

In this section, we evaluate ROPFuscator by addressing the following research questions.

- **RQ1: Completeness**. What is the maximum code coverage this methodology can achieve?
- **RQ2: Performance**. To what extent this obfuscation technique affects performance?
- **RQ3: Correctness**. Are the semantics of the program preserved?
- **RQ4: Robustness**. How is robustness of the obfuscation mechanism in regards to threat model attacks?
- **RQ5: Practicality**. Is our approach applicable to real-world use cases?

We address the first question by measuring the coverage of instructions we successfully obfuscate in §4.1, and answer RQ2 by measuring obfuscated code's performance in §4.2.

Table 2: Ratio of instructions obfuscated in SPEC CPU 2017 (SPECrate Integer) C/C++ test cases

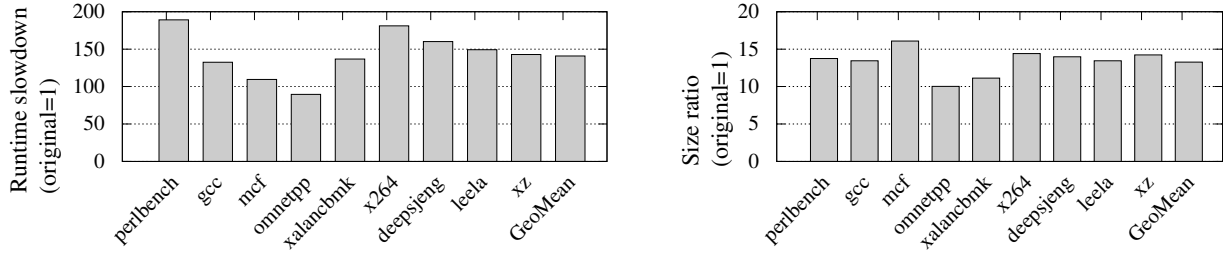| Option	Status	perlbench	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	xz	W.AVG
-O0	Obfuscated	74.16%	76.67%	64.79%	65.28%	66.03%	64.75%	68.18%	68.51%	66.23%	72.20%
	Unobfuscated (No gadget / reg)	9.83%	8.59%	11.68%	7.88%	8.31%	13.50%	11.38%	6.59%	11.86%	8.90%
	Unobfuscated (Other)	16.01%	14.74%	23.53%	26.84%	25.66%	21.75%	20.44%	24.90%	21.91%	18.90%
-O3	Obfuscated	40.41%	42.41%	29.08%	43.20%	39.34%	26.89%	33.14%	42.29%	32.02%	40.33%
	Unobfuscated (No gadget / reg)	10.50%	7.71%	13.33%	7.41%	9.53%	11.22%	13.33%	9.47%	11.73%	8.74%
	Unobfuscated (Other)	49.09%	49.88%	57.59%	49.38%	51.14%	61.89%	53.54%	48.24%	56.25%	50.93%

Figure 4: Runtime slowdown and code size of obfuscated programs for SPEC CPU 2017 (SPECrate Integer)

During the test runs, we also observe if the program is executed correctly (RQ3). Then we evaluate robustness (RQ4) against attacks (defined in §2.1) in §4.3. Finally, we discuss practicality (RQ5) with a case study applying it to an open-source program in §4.4.

We evaluate several obfuscation configurations throughout the section. We use words 'ROPonly', 'ROP+OP$_{Basic}$', 'ROP+OP$_{DSE}$', 'ROP+OP$_{DSE}$+Hiding' to denote ROP transformation (§2.2.1) only, ROP transformation + basic opaque predicates (§2.2.2), ROP transformation + DSE-resistant opaque predicates, and ROP transformation + opaque predicates + instruction hiding (§2.2.3) respectively. We also use 'Baseline' for non-obfuscated binary.

Test Sets and Experiment Environment: We evaluate coverage, execution speed (throughput), and code size in two test sets: 1) SPEC CPU2017 (SPECrate Integer) C/C++ tests 2) custom workload using binutils. We evaluated the robustness with a custom simple program that behaves like a program. Unless explicitly stated, the following evaluations are done with the following conditions: the program is compiled with optimization option -O0 (no optimization), and gadgets are extracted from 32bit libc version 2.27-3ubuntu1. Programs are executed in Ubuntu 18.04 x86-64. We acknowledge that an evaluation in a virtual environment is affected by VMM overhead and noisy neighbors, but as described below, the numbers are about 10-1000x, and we are only interested in the order of magnitude so that the deviation would be negligible.

4.1 Completeness: Obfuscation Coverage

We first evaluate a ratio of instructions obfuscated by ROP transformation. Since static disassemblers translate binary code into assembly, it cannot directly disassemble the translated ROP chain into the original code. Thus ROPFuscator is more robust if more instructions are transformed.

Table 2 shows ratio of obfuscated instructions in SPEC CPU test cases, with optimization options -O0 (not optimized) and -O3 (highly optimized). The result shows that with optimization option -O0, around 60–80% of the instructions can be obfuscated into ROP chains. On the other hand, only 40% instructions on average can be obfuscated when compiled with -O3 option. Since x86 is a Complex Instruction Set Computer (CISC) architecture, optimization can turn a program into more complex instructions that are hard to express as a combination of simple ROP microgadgets. This result suggests that it is better to use -O0 when applying this obfuscation mechanism. Although this causes performance degradation, performance loss due to obfuscation is much larger than the optimization option's loss. We discuss about performance further in §4.2.

About 7–12% of the total instructions were not obfuscated in both cases because no free registers or gadgets are available. We believe that this would not be practically a problem since the attacker cannot understand the meaning of the exposed instruction solely without knowing the rest of the instructions. Also, we would be able to obfuscate these instructions by 1) saving and restoring registers to allocate free temporary registers and 2) create our own library that includes gadgets.

Next, we observe the obfuscation coverage difference due to library versions. Table 3 shows the ratio of instructions obfuscated for two programs in binutils 2.32. With libc version 2.27-3ubuntu1 (on Ubuntu 18.04), the result shows similar numbers to SPEC CPU (70–80%), but if we use libc version 2.27-3ubuntu1.2, numbers decrease to 20–40%, while no gadget error raises to about 50–60%. We investigated the reason,

Table 3: Ratio of instructions obfuscated in binutils for different libc versions

libc version	Status	readelf	c++filt
2.27-3 ubuntu1	Obfuscated	77.24%	74.99%
	Unobfuscated (No gadget / reg)	11.80%	11.70%
	Unobfuscated (Other)	10.96%	13.31%
2.27-3 ubuntu1.2	Obfuscated	36.02%	26.07%
	Unobfuscated (No gadget / reg)	53.02%	60.62%
	Unobfuscated (Other)	10.96%	13.31%
2.31-0 ubuntu9	Ofuscated	82.69%	80.93%
	Unobfuscated (No gadget / reg)	6.35%	5.77%
	Unobfuscated (Other)	10.96%	13.31%

Table 4: Runtime slowdown and code size of obfuscated programs for binutils for each obfuscation algorithm

metric	obfuscation	absolute value		ratio (Baseline=1)		ratio (Roponly=1)	
		readelf	c++filt	readelf	c++filt	readelf	c++filt
time	Baseline	0.39s	0.30s	1.0	1.0	0.09	0.01
	ROPonly	4.23s	30.6s	11.0	102	1.0	1.0
	ROP+OP$_{Basic}$	41.1s	337s	107	1118	9.7	11.0
	ROP+OP$_{DSE}$	66.4s	761s	172	2527	15.7	24.8
	ROP+OP$_{DSE}$+Hiding	57.1s	611s	148	2030	13.5	19.9
size	Baseline	1.1MB	1.1MB	1.0	1.0	0.10	0.07
	ROPonly	10.5MB	15.7MB	9.6	14.1	1.0	1.0
	ROP+OP$_{Basic}$	895MB	1407MB	828	1269	86.6	89.7
	ROP+OP$_{DSE}$	1530MB	2411MB	1417	2175	148	154
	ROP+OP$_{DSE}$+Hiding	1283MB	2063MB	1188	1861	124	132

and the reason was the second library does not have gadget `xchg eax, edx; ret`. As described earlier, we convert single instruction to a combination of microgadgets [18], and in this process, we heavily rely on exchange (`xchg`) gadgets. If the above gadget is not available, we cannot exchange `edx` and other registers, and many other microgadgets which involve `edx` register cannot be used unless the original code is exactly using `edx` register. This result shows that ROP transformation is very sensitive to ROP gadget availability, so we should carefully choose which library we use for gadget extraction. However, if we can choose library version arbitrarily, the ratio of no gadgets and registers error can be as low as 5–7% (using libc version 2.31-0ubuntu9). Alternatively, we can use a custom library where all required gadgets are embedded into, as explained above.

> **Takeaway—RQ1: Completeness**.
>
> On average, ROPFuscator obfuscates about 60–80% of the instructions with ROP transformation. The number depends on the compiler optimization option and shows better coverage without optimization. The number also depends largely on the library version from which the ROP gadgets are extracted, and selecting an appropriate library version ensures high coverage.

4.2 Performance and Correctness

Secondly, we evaluate run-time slowdown and code size bloat up introduced by ROPFuscator. We obfuscate the entire SPEC CPU 2017 benchmark programs as well as some of the binutils programs with optimization disabled and compare the result with original programs.

The result for SPEC CPU is shown in Figure 4. With ROP transformation, execution time is about 140x longer on the geometric mean (109–189x), and size is 13x larger on weighted average (10–16x). It takes too long to run SPEC CPU test cases obfuscated with opaque predicates, so we evaluate performance and code size with custom workload with binutils. The result in Table 4 shows that opaque predicate obfuscation increases execution time by about 10x (without DSE counter-

measure) or 20x (with DSE countermeasure), and code size by about 90x (without DSE countermeasure) or 150x (with DSE countermeasure), compared to ROP transformation only. Instruction hiding decreases the number of ROP gadgets and performs slightly better (execution time is about 16x longer, size is about 130x larger than ROP transformation only).

According to both measurements, the execution time will be 10–200x with ROP transformation only, 200–4000x with ROP together with DSE-resistant opaque predicates, and 150–3000x with full obfuscation. The executable size will be 10–16x with ROP transformation only, 1500–2500x with ROP and DSE-resistant opaque predicates, and 1200–2000x with full obfuscation.

We would also like to note that we do not observe any behavioral differences with the original programs during the above run. We designed every obfuscation transformation very carefully as described in §3.2. Though it is not formally proven to be correct, these test programs are large and practical, and the result is enough to convince us that ROPFuscator can preserve program semantics accurately.

> **Takeaway—RQ2-RQ3: Performance and Correctness**.
>
> ROP transformation imposes about 10–200x of execution time overhead and 10–15x of code size overhead. DSE-resistant Opaque predicates imposes further 20x (total 200–4000x) of execution time overhead and further 150x (total 1500–2500x) of code size overhead. These numbers may seem huge, but it is possible to control performance loss by choosing where to strongly obfuscate, as discussed later in §4.4 and §5. A summary is shown in Table 6 together with robustness evaluation.
>
> ROPFuscator preserves original semantics of programs throughout obfuscation transformations.

4.3 Robustness

We evaluate robustness with respect to the threat model defined in §2.1. To make the discussion clearer, we use two variations of simple input validation programs shown in Figure 5. They are intended as a simpler form of product code

(a) early-exit function   (b) late-exit function

```c
int check(const char *s) {        int check(const char *s) {
                                    int i = 0;
  if (s[0] != 'H') return 0;        i += (s[0] == 'H');
  if (s[1] != 'e') return 0;        i += (s[1] == 'e');
  if (s[2] != 'l') return 0;        i += (s[2] == 'l');
  ...                               ...
  if (s[12] != '\0') return 0;      i += (s[12] == '\0');
  return 1;                         return i == 13;
}                                 }

            int main(int argc, char **argv) {
              if (check(argv[1])) puts("OK");
              return 0;
            }
```

Figure 5: Example source code to be obfuscated, which implements input validation

checking function.

In this section, we evaluate each threat model defined in §2.1: threat A (decompiler), B (static ROP analysis), C (dynamic symbolic execution) and D (dynamic ROP analysis).

A: Decompilation   We evaluate robustness against Threat A: decompilers by decompiling simple function listed in Figure. 5 with or without obfuscation and comparing the results. We used open-sourced state-of-the-art decompilers to reverse-engineer binaries: Ghidra[1], retdec[2] and r2dec[3]. As a result, all decompilers can reconstruct functions without obfuscation, but none of them reconstruct the original code structure with ROP transformation. This is likely because the decompilers do not understand the ROP chain structure nor correctly recognize function boundary. Even though some of them can decompile opaque predicates, the analysis requires extensive human intervention.

B: Static ROP Chain Analysis   Here we evaluate the robustness of ROPFuscator against Threat B: static ROP chain analysis.

First, we implemented a deobfuscator to convert the ROP chain back to the original code, using a similar technique to deRop [15]. We identify ROP gadgets and combine underlying code into original instructions. Instead of directly processing ROP chain in memory, our deobfuscation approach statically detects ROP chain generator code by looking for stack manipulating instructions (push, pop and ret). Then we simulate the stack content and locate ROP gadgets and concatenate the underlying instructions.

We applied this deobfuscator to binaries obfuscated with two configurations: ROPonly and ROP+OP$_{DSE}$. We obfuscated the early-exit function shown in Figure. 5 (a), applied our deobfuscator, and decompiled with Ghidra. When deobfuscated ROPonly binary, the recovered instructions are slightly different from the original code, but Ghidra successfully recovered almost the same control structure as the orig-

---

inal C source code. On the other hand, ROP+OP$_{DSE}$ binary cannot be deobfuscated by this approach since opaque predicates use many kinds of instructions that our deobfuscator cannot handle. This result shows that the ROP transformation itself can be broken by static ROP chain analysis but can be hardened by opaque predicates.

Secondly, we analyze immediate operands that appear in ROP chains. In the example shown in Figure. 5, byte comparison instructions `cmp eax,0x48; cmp eax,0x65; ...` are used periodically. Those instructions are converted to a pattern like `push 0x48; push G(pop ecx); ...` in ROP chain, and by looking at pushed constant, it is possible to extract string constant. We wrote a script to automate extracting data from these patterns (periodic occurrence of immediate operands with various intervals) and successfully recovered string constant in both non-obfuscated binary and obfuscated binary (ROPonly). On the other hand, we could not recover immediate operands from binary obfuscated with opaque predicates (ROP+OP$_{DSE}$). This result shows that it is important to protect immediate operands using opaque predicates.

C: Dynamic Symbolic Execution   As noted in §2.1, we consider Threat C: DSE for an input finding attack which computes input value that passes the validation functions. We created a script based on angr to find a possible input that will cause the program shown in Figure. 5 to output 'OK'.

We applied this script to programs with each obfuscation configuration. We also change some DSE (angr) execution strategy parameters: depth-first vs. breadth-first search and symbolic vs. tracing. We ran the script with a memory threshold of 8GB and measured the time and memory needed to compute the input. The result is shown in table 5.

The result shows that DSE can crack non-obfuscated (Baseline) and ROPonly binaries within 10 seconds and ROP+OP$_{Basic}$ binary in 1 minute, while it cannot crack ROP+OP$_{DSE}$ and ROP+OP$_{DSE}$+Hiding binaries because of memory overflow. It shows that ROP, combined with DSE-resistant opaque predicates, will almost nullify input finding attacks with DSE, even for a straightforward program like this. Applying ROP solely or using opaque predicates without user input is not a good defense against DSE; it does not increase analysis time/memory very much compare to its runtime slowdown.

This experiment also shows the effectiveness of using user input value as input to opaque predicates; we looked into the compiled example code and found that each input byte goes into eax register when comparing it against an expected value. This verifies our intuition that registers contain user input, suggested in 3.3.

D: Dynamic ROP Chain Analysis   Lastly, we evaluate the robustness of ROPFuscator against Threat D: dynamic ROP chain analysis. As noted in the previous experiment, static analysis has a limitation that we cannot analyze all

9

Table 5: DSE analysis time and consumed memory for different obfuscation configurations and exploration strategies

| Program | Obfuscation config | DSE exploration strategy in angr | | | | | | | |
| | | Symbolic/BFS | | Symbolic/DFS | | Tracing/BFS | | Tracing/DFS | |
		Time	Memory	Time	Memory	Time	Memory	Time	Memory
Early-exit	Baseline	5.4s	170MB	5.5s	173MB	4.5s	170MB	4.4s	169MB
	ROPonly	9.5s	168MB	8.0s	164MB	9.5s	173MB	7.4s	176MB
	ROP+OP$_{Basic}$	85.3s	417MB	57.4s	365MB	85.7s	413MB	56.1s	368MB
	ROP+OP$_{DSE}$	Out of Memory		Out of Memory		Out of Memory		Out of Memory	
	ROP+OP$_{DSE}$+Hiding	Out of Memory		Out of Memory		Out of Memory		Out of Memory	
Late-exit	Baseline	4.5s	130MB	4.5s	130MB	3.7s	130MB	3.7s	130MB
	ROPonly	7.1s	138MB	7.3s	134MB	7.0s	141MB	7.0s	141MB
	ROP+OP$_{Basic}$	69.7s	324MB	68.2s	326MB	74.1s	342MB	74.0s	345MB
	ROP+OP$_{DSE}$	Out of Memory		Out of Memory		Out of Memory		Out of Memory	
	ROP+OP$_{DSE}$+Hiding	Out of Memory		Out of Memory		Out of Memory		Out of Memory	

Out of Memory: force stopped after exceeding 8000MB

opaque predicates' instructions. We used an approach similar to [16, 21] to emulate the ROP chain building code to implement a dynamic ROP deobfuscator. Given a code range, the deobfuscator executes the code range with a CPU simulator and collects the execution trace.

We applied this dynamic ROP deobfuscator to the early-exit function shown in Figure. 5 (a) to both ROP+OP$_{DSE}$ and ROP+OP$_{DSE}$+Hiding. This deobfuscator successfully extracted the code which is semantically equivalent to the original one, from ROP+OP$_{DSE}$ obfuscated binary. However, it can only extracted partial code from ROP+OP$_{DSE}$+Hiding obfuscated binary. For example, `mov ecx,0x48; xchg eax,edx; mov eax,edx; sub eax,ecx; xchg eax,edx` are extracted from ROP+OP$_{DSE}$ binary, while only `mov eax,edx; sub eax,ecx; xchg eax,edx` are extracted from ROP+OP$_{DSE}$+Hiding binary, which means the first two instructions are hidden in the opaque predicates. This means that opaque predicates are not robust against the dynamic tracing attack, but instruction hiding can be used to prevent the attack from revealing the entire code.

Takeaway—RQ4: Robustness.

ROP transformation is robust against threat A (disassembler and decompiler) but not robust against threat B (static ROP), C (DSE), and D (dynamic ROP). Introducing opaque predicates fortifies obfuscated programs against B (static ROP) and C (DSE). Introducing instruction hiding further inhibits full disclosure of code by threat D (dynamic ROP), making it hard to decompile the obfuscated binary to the original code. A summary is shown in Table 6.

4.4 Practicality: Case Studies

In §4.2, we evaluated the performance of ROPFuscator with hypothetical workloads, and the impact was significant. Such overhead is not compatible with any real-world application of our obfuscation technique. For this reason, we considered alternative options that retained a better performance overhead. We assume that functions that produce or operate on sensitive data and, therefore, subject to obfuscation interest take up a small portion of the total execution time. To balance robustness and performance, we considered to selectively obfuscate such functions and we later verify our assumptions in the rest of this section.

We apply ROPFuscator to two widespread use cases: multimedia copyright protection and software license protection. We explain how obfuscation mechanisms can be applied to protect critical assets in the program, balancing robustness and performance.

4.5 Multimedia Copyright Protection

Digital rights management (DRM) [32] is a mechanism to protect commercial media content such as DVD videos against digital piracy. It uses an encryption (or scrambling) mechanism to protect media content, but attackers try to retrieve the keys to decrypt the content to bypass protection and copy the material illegally. Obfuscation has played an important role in preventing attackers from retrieving the keys by reverse engineering.

First, we adopt ROPFuscator to an open-source video player VLC Media Player[4] together with DVD descrambling library libdvdcss[5]. Upon VLC Media Player's request, libdvdcss library derives the content decryption key (title key) from the protected DVD media and decrypt (descramble) the DVD content, which is further decoded by VLC Media player to be played on-screen.

Though the final goal is to protect media content, it is more important to protect the title keys and key derivation pro-

---
[4] https://www.videolan.org/vlc/
[5] https://www.videolan.org/developers/libdvdcss.html

Table 6: Robustness and performance of each algorithm in ROPFuscator against attacks

Obfuscation Algorithm	Robustness against Attack Algorithm				Performance	
	A) Decompiler	B) Static ROP	C) DSE	D) Dynamic ROP	Slowdown ratio	Size ratio
Baseline	○	○	○	○	1	1
ROPonly	●	○	○	○	10–200	10-16
ROP+OP$_{Basic}$	●	●	○	○	100–2000	900–1500
ROP+OP$_{DSE}$	●	●	●	○	200–4000	1500–2500
ROP+OP$_{DSE}$+Hiding	●	●	●	◑	150–3000	1200–2000

○: Breakable, ●: Robust, ◑: Mostly Robust

Table 7: Execution time, code size and behavior of VLC media player using libdvdcss obfuscated with various options

Config	Time [s]	CPU Usage	Played Smoothly?	Size [MB]
Baseline	30.2	12.4%	Yes	0.034
ROPonly	30.2	23.3%	Yes	0.38
ROP+OP$_{DSE}$	110.2	97.0%	No	48.5
ROP+OP$_{DSE}$+Hiding	120.7	95.3%	No	41.3
Balanced	30.2	23.2%	Yes	18.4

Table 8: Execution time and code size of a License++ program obfuscated with various options

Obfuscation	Time[s]	Size[MB]
Baseline	0.005	3.2
ROPonly	0.055	17.0
ROP+OP$_{DSE}$	37.0	2157
ROP+OP$_{DSE}$+Hiding	29.1	1866
Balanced	0.158	115

cess. Therefore, we prioritized the obfuscation of title key derivation functions than content decryption functions in the balanced configuration below.

We obfuscate libdvdcss with four configurations and compare performance: 1) obfuscating all function with ROP only (ROPonly) 2) obfuscate all functions with ROP + opaque predicates (ROP+OP$_{DSE}$) 3) obfuscate all functions with ROP + opaque predicates and instruction hiding (ROP+OP$_{DSE}$+Hiding) 4) obfuscate title key derivation functions with ROP+OP$_{DSE}$+Hiding and the rest of the library with ROPonly (Balanced). We identified title key derivation functions with function name containing `key` or `Key`, and function `AttackPattern`.

Then we ran VLC media player to play a commercial DVD title for 30 seconds, with each libdvdcss.so loaded. The performance result is shown in Table 7. In the table, Baseline shows the result without obfuscation, and the other three rows show the time, average CPU usage, and if it played smoothly. The average CPU usage is calculated by dividing CPU utilization time by total execution time, i.e. $(T_{user} + T_{system})/T_{real}$. Without obfuscation, the program plays DVD video smoothly, and CPU usage is around 10%. When we apply ROP transformation, it still plays video smoothly, but CPU usage is raised by about 10 %points. If we apply opaque predicate upon ROP transformation, the decryption process is not in time for real-time playing, and the player frequently stops to buffer movie content, showing CPU usage at almost 100%. When we obfuscate the library with a balanced configuration, it plays almost in the same performance as ROP transformation only.

This result seems to support our assumption: a confidentially critical part of the program is not executed many times, and obfuscating it does not have a great performance impact.

We measured the number of function calls and instructions using Valgrind [33]. libdvdcss accounts for about 5.5% of total instructions executed. Among libdvdcss, DVD decryption function (`dvdcss_unscramble`) accounts for 99.88% of instructions (15091 out of 15970 function calls), while key derivation functions only account for 0.028% of instructions (63 function calls). Using slower (more robust) obfuscation for only 0.0015% of overall instructions (0.028% of libdvdcss) does not greatly impact performance; total overhead would be about 6% even if execution speed overhead is 4000x.

4.6 Software License Protection

We test software license protection use case with an open-source program License++[6]. It utilizes an open-source C++ cryptography library to verify the cryptographic signature of a license file. We obfuscate the license check algorithm together with encryption algorithms to see the entire performance.

The result is shown in Table 8. In 'opaque' and 'full' configuration, most overhead in execution time is caused by the program loading process since the executable program size is about 2GB, and there are many relocations in the executable. This means that we should shrink executable size for practical use. Since most of the code consists of unused functions in the cryptographic library, we apply strong obfuscation only to the entire license checking algorithm and related cryptographic functions. As a result, the overhead is about 0.15 seconds in execution time and 112MB in code size. We believe less than 0.5 seconds of overhead within startup time is negligible.

---

[6] https://github.com/amrayn/licensepp

> **Takeaway—RQ5: Practicality**.
>
> We applied `ROPFuscator` to two real-world typical use cases, protected multimedia player, and software license verifier. It is possible to select sensitive functions which should be strongly protected to achieve robustness for acceptable performance loss. We also give examples of how we prioritize such functions to balance robustness and performance.

5   Discussion

In the previous section, we evaluated our work's performance, robustness, coverage, and correctness and demonstrated a real-world application of our technique.

We also demonstrated that `ROPFuscator` is robust against modern reverse engineering methodologies as defined in the threat model and, on average, can protect 70% of the code (specifically, 60%–80% according to §4.1). Our experiments highlighted the trade-off between the performance and robustness of our approach (Table 6). However, this can be balanced on a per-function basis by tuning the obfuscation layers (§4.4).

Although `ROPFuscator` is not specifically designed for data obfuscation, it can be used to protect constant values by leveraging opaque constants (§2.2.2). Moreover, users can embed and protect sensitive data such as constants used in whitebox cryptography [34].

Next, we have evaluated our method against other obfuscation techniques.

Resistance to Other Obfuscation Approaches   There are several reverse engineering techniques which have been proposed recently: opaque predicate identification [29, 30], virtualization [35] and program-synthesis [36] deobfuscation. This section discusses the applicability of these methodologies against our model and how our approach performs countering each of them.

First, we implemented a countermeasure to opaque predicate identification attacks [29, 30]. We could not test the aforementioned works for different reasons. We encountered technical issues in compiling the project by Ming et al. [29] that promptly offered us support. Unfortunately, the issues persisted. Differently, we could not evaluate against the work by Ming et al. [29] due to the code being unavailable. For these reasons, we designed the opaque predicates so that their output cannot be easily inferred even if identified (§3.3.2).

Later, we evaluated a virtualization deobfuscation approach, VMHunt [35]. Xu et al. claim their method is also compatible with ROP chains since they consider gadgets as VM instructions. Unfortunately, VMHunt was unable to identify any ROP gadgets in our experiments.

Finally, we tested our technique against Syntia, a deobfuscation framework based on program synthesis. Syntia synthesizes the semantics of a program from its input-output relation. According to our experiments, it can recover the semantics of simple functions (e.g. $f(x, y) = 2x - y$), but regardless of obfuscation, it could not recover more complex functions (e.g. $f(x, y) = 4x + y$). Considering that the functions we encountered in our experiments were more complex than the aforementioned examples, Syntia could not properly address our approach.

A common dynamic analysis task is to debug a process; therefore, we considered a scenario where the attacker has complete knowledge of `ROPFuscator` and its mechanics. In this case, we are aware that it is impossible to inhibit an attacker from setting a breakpoint before a `ret` instruction to then single step and trace the ROP chain execution. However, instruction hiding (§2.2.3) offers some protection by corrupting the execution trace, which satisfies our goal of increasing the analysis time cost.

6   Related Work

This section briefly discusses the related studies on obfuscation and ROP. Moreover, we explain their relation to our approach.

ROP-based obfuscation   ROP is applied in various ways to protect software. The most common is its application by malware authors to evade detection from signature-based software such as anti-virus solutions. Frankenstein [11] extracts ROP gadgets from benign binaries and combines them to generate ROP chains that execute malicious actions. However, this approach is not robust enough in countering an attack in a MATE scenario since it is not designed to withstand targeted ROP analyses. ROP needle [12] uses a similar technique to evade anti-virus detection by encrypting and decrypting the ROP chains on-the-fly using an externally supplied encryption key. ROP needle fits more a malicious scenario where malware authors intend to protect their work from analysts for a determined time-frame (e.g., the duration of a malware campaign). However, there is no specified time limit for reverse engineering in commercial software protection, exposing the encryption key to an eventual disclosure to malicious analysts.

Another application is software tampering. Parallax [37] proposes a mechanism to embed ROP gadgets into sensitive code regions. Modifying these code regions leads the ROP chain to be corrupted hence impeding its proper execution. This can deceive debuggers when setting software breakpoints since they inherently change the program code. Therefore, Parallax focuses on protecting software integrity and not its confidentiality.

In conclusion, several mechanisms protect software in MATE scenarios, sharing the objectives defined in our

work [13, 38]. RopSteg [38] proposes an instruction steganography algorithm. It injects ROP chains in code regions along with extra bytes. As an effect, this causes disassemblers to disassemble instructions erroneously. However, this considers only static analyses (Threat A) hence excluding a targeted ROP chain analysis executed through dynamic tracing (Threat D). ROPOB [13] exclusively obfuscates the control flow of a program, leaving non-control instructions unobfuscated, and its threat model does not consider targeted ROP analyses (Threat B and D).

ROP Chain Generation Q [24] proposes an approach to generate ROP chains. It defines semantic operations for branches, memory load/store, and arithmetic calculations. Later, it extracts and lifts the gadgets into an intermediate language that is finally compiled to a ROP chain as a result.

This technique is effective in generating ROP chains. We share this trait due to our ROP transformation pass, although its final goal is not obfuscation and thus has no discussion about preventing reverse engineering.

Opaque Predicates and Opaque Constants We use opaque predicates based on previous work. There is a multitude of algorithms proposed to generate them, that span across various calculations including arithmetic operations, non-determinism [6], one-way functions [26], and computationally hard problems (e.g. pointer-aliasing [6] or 3SAT [19, 27]). Furthermore, three main types of opaque predicates are documented in literature: invariant, contextual, and dynamic opaque predicates. *Invariant* opaque predicates always evaluate to the same value, decided a-priori by the obfuscator. *Contextual* opaque predicates change their output based on preconditions chosen at design-time [31]. *Dynamic* opaque predicates introduce the idea of correlated predicates that maps the output of a predicate to the input of subsequent one [39].

These proposals are diagonal to our approach, i.e., we can enhance ROPFuscator by integrating them as a component in our obfuscation. For this reason, we considered the use of opaque constants to compute gadget addresses [19].

7   Conclusion

Software obfuscation techniques are evolving to challenge new reverse engineering techniques. Inspired by the advantage of return oriented programming which interferes with our natural perception of program execution, we present ROPFuscator, a framework for fine-grained obfuscation of C/C++ programs based on ROP (§3.2). Although previous work already explores the effectiveness of ROP as an obfuscation technique (§6), our approach deals with evolving reverse engineering attacks by introducing a unified threat model for ROP-based obfuscation techniques (§2.1). We introduce a novel instruction hiding technique (§3.4), later integrated with opaque predicates and constants (§3.3), to provide a configurable yet robust framework.

The arms race between software obfuscation and reverse engineering seems to be endless. We introduce an unified threat model and a thorough evaluation along multiple dimensions (§4) to help us reason about the decisions involved in designing or choosing obfuscation techniques. This is an effort to provide researchers and practitioners a better understanding of strengths and limitations of obfuscation mechanisms (§2.2.1-2.2.3).

References

[1] David Aucsmith. Tamper resistant software: an implementation. In *Proc. IH '96*, pages 317–333, 1996.

[2] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. ISC '01*, pages 144–155, 2001.

[3] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. CCS '03*, pages 290–299, 2003.

[4] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1), 2016.

[5] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. COMPSAC '03*, pages 170–179, 2003.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. POPL '98*, pages 184–196, 1998.

[7] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proc. DRM '06*, pages 47–58, 2006.

[8] Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh. Software protection through anti-debugging. *IEEE Secur. Priv.*, 5(3):82–84, 2007.

[9] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. CCS '07*, pages 552–561, 2007.

[10] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[11] Vishwath Mohan and Kevin W Hamlen. Franken-stein: Stitching malware from benign binaries. In *Proc. WOOT '12*, 2012.

[12] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. The ROP needle: hiding trigger-based injection vectors via code reuse. In *Proc. SAC '19*, pages 1962–1970, 2019.

[13] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao, and Lei Shi. ROPOB: Obfuscating binary code via return oriented programming. In *Proc. SecureComm '17*, volume 238, pages 721–737, 2018.

[14] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *J. Netw. Comput. Appl.*, 48:44–57, 2015.

[15] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. deRop: removing return-oriented programming from malware. In *Proc. ACSAC '11*, pages 363–372, 2011.

[16] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *Proc. ASIACCS '16*, pages 47–58, 2016.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *Proc. S&P '16*, pages 138–157, 2016.

[18] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *Proc. WOOT '12*, 2012.

[19] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proc. ACSAC '07*, pages 421–430, 2007.

[20] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proc. ACSAC '19*, pages 177–189, 2019.

[21] Daniele Cono D'Elia, Emilio Coppa, Andrea Salvati, and Camil Demetrescu. Static analysis of ROP code. In *Proc. EuroSec '19*, pages 1–6, 2019.

[22] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proc. ACSAC '16*, pages 189–200, 2016.

[23] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing resilient bi-opaque predicates against symbolic execution. In *Proc. DSN '18*, pages 666–677, 2018.

[24] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proc. USENIX Security '11*, 2011.

[25] Ariel Tamches and Barton P Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. OSDI '99*, 1999.

[26] Lukas Zobernig, Steven D. Galbraith, and Giovanni Russello. When are opaque predicates useful? In *Proc. TrustCom/BigDataSE '19*, pages 168–175, 2019.

[27] Brendan Sheridan and Micah Sherr. On manufacturing resilient opaque constructs against static analysis. In *Proc. ESORICS '16*, volume 9879, pages 39–58, 2016.

[28] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[29] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proc. CCS '15*, pages 757–768, 2015.

[30] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proc. SPRO'19*, pages 3–14, 2019.

[31] Stephen Drape. Intellectual property protection using obfuscation. Technical Report CS-RR-10-02, University of Oxford, 2010.

[32] Qiong Liu, Reihaneh Safavi-Naini, and Nicholas Paul Sheppard. Digital rights management for content distribution. In *Proc. ACSW Frontiers '03*, volume 21, pages 49–58, 2003.

[33] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI '07*, pages 89–100, 2007.

[34] Yuan Xiang Gu, Harold Johnson, Clifford Liem, Andrew Wajs, and Michael J. Wiener. White-box cryptography: practical protection on hostile hosts. In *Proc. SSPREW '16*, pages 1–8, 2016.

[35] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proc. CCS '18*, pages 442–458, 2018.

[36] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *Proc. USENIX Security '17*, 2017.

[37] Dennis Andriesse, Herbert Bos, and Asia Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In *Proc. DSN '15*, pages 125–135, 2015.

[38] Kangjie Lu, Siyang Xiong, and Debin Gao. RopSteg: program steganography with return oriented programming. In *Proc. CODASPY '14*, pages 265–272, 2014.

[39] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *Proc. ACSAC'00*, pages 308–316, 2000.

A Listings of Obfuscated Code

In this appendix, we exemplify how a program is transformed in more detail. Throughout the section, we use a simplified variant of the program shown in Figure 3 as an example:

```
int check(const char *c) {
  if (s[0] == 'A') return 0;
  if (s[1] == 'B') return 0;
  if (s[2] == 'C') return 0;
  if (s[3] != '\0') return 0;
}
```

Compiled Program (Unobfuscated)   Figure 6 shows unobfuscated program: a) is a (disassembled) view of the program compiled to x86 instructions, b) is a decompiled code by Ghidra and c) is the control flow graph generated by Ghidra. The code a) is easily understood by software engineers who knows x86 instruction set, and b) is equivalent to the original code.

ROP Transformation   Figure 7 shows the program after ROP transformation (ROPonly). The disassembled code shown in a) contains many push instructions followed by ret instruction. Each pushed address is expressed as an offset from a random libc function symbol. b) and c) shows the decompiled code and control flow using Ghidra 9.1.2. Ghidra sequentially analyzes the function from the entry point, and it stops analysis when it reaches the first ret instruction without any jumps. Therefore it displays only a single block (the first ROP chain builder code) as the entire function flow. It does not compile the code correctly, since only push instructions are executed and they do not affect return values (normally stored in eax register).

Opaque predicates   Figure 9 shows the program obfuscated by ROP+OP$_{Basic}$. Similar to the ROPonly case, Ghidra only analyzes instructions before the first ret. Ghidra optimizes away the opaque predicates from the decompiled code, since it does not affect eax when it reaches ret. The control flow is a single block, similarly to ROPonly but with much more instructions in the block.

Instruction Hiding   Figure 10 shows the program obfuscated by ROP+OP$_{DSE}$+Hiding. The disassembled code is mostly the same as that of ROP+OP$_{Basic}$ except that hidden instructions and dummy instructions are inserted in-between opaque predicates. Decompiled code and control flow are almost the same as ROP+OP$_{Basic}$ and omitted here.

Deobfuscation with Static ROP Chain Analysis   Figure 8 shows the program deobfuscated by Threat B: static ROP chain analysis in the course of evaluation in §4.3. Although disassembled code has redundant instructions (e.g. xchg) and replaced instructions (e.g. cmp with sub), the decompiled code and control flow look similar to the original ones shown in Figure 6. We confirmed that static ROP chain analysis is not applicable to programs obfuscated by opaque predicates, so the deobfuscation result is not listed here.
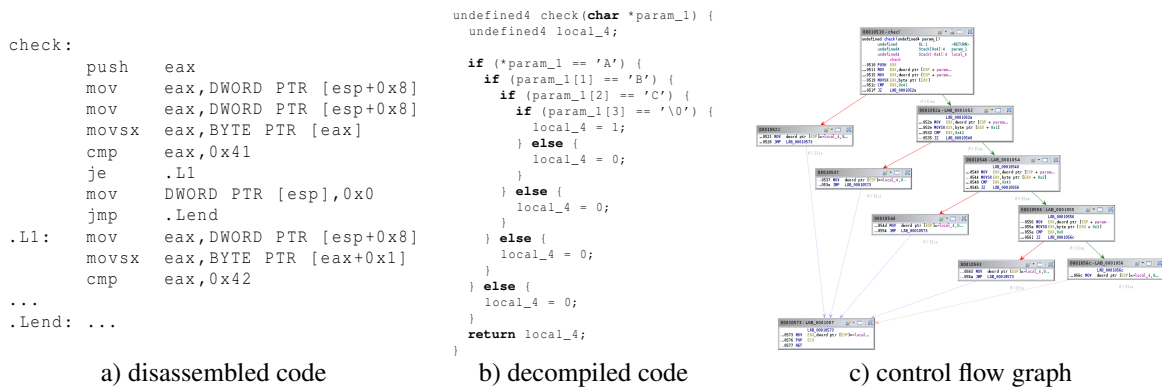
```
check:
        push    eax
        mov     eax,DWORD PTR [esp+0x8]
        mov     eax,DWORD PTR [esp+0x8]
        movsx   eax,BYTE PTR [eax]
        cmp     eax,0x41
        je      .L1
        mov     DWORD PTR [esp],0x0
        jmp     .Lend
.L1:    mov     eax,DWORD PTR [esp+0x8]
        movsx   eax,BYTE PTR [eax+0x1]
        cmp     eax,0x42
...
.Lend: ...
```

a) disassembled code

```c
undefined4 check(char *param_1) {
  undefined4 local_4;

  if (*param_1 == 'A') {
    if (param_1[1] == 'B') {
      if (param_1[2] == 'C') {
        if (param_1[3] == '\0') {
          local_4 = 1;
        } else {
          local_4 = 0;
        }
      } else {
        local_4 = 0;
      }
    } else {
      local_4 = 0;
    }
  } else {
    local_4 = 0;
  }
  return local_4;
}
```

b) decompiled code

c) control flow graph

Figure 6: Disassembled / decompiled view of original program

```
...
1eec:   push    0x1f0d                              # jump to L1
1ef1:   push    __strspn_c1+0x644a6                 # xchg eax,edx; ret
1ef6:   push    _IO_switch_to_get_mode+0xdc4a8      # sub eax,ecx; ret
1efb:   push    strerror_r+0x10aba                  # mov eax,edx; ret
1f00:   push    _IO_getline+0x82bf6                 # xchg eax,edx; ret
1f05:   push    0x41                                # (value 0x41)
1f07:   push    vprintf+0x14a664                    # pop edx; ret
1f0c:   ret
1f0d:   lea     esp,[esp-0x18]                      # L1
1f11:   pushf
1f12:   lea     esp,[esp+0x1c]
1f16:   push    xdr_int32_t+0xfff00c4b              # push eax; ret
1f1b:   push    vasprintf+0x76b43                   # cmove eax,ecx; ret
1f20:   push    0x1f3a                              # jump to L2
1f25:   push    _IO_wfile_underflow+0xfffef445      # pop eax; ret
1f2a:   push    0x1f5e                              # jump to L3
1f2f:   push    _IO_wfile_underflow+0x14c72a        # pop ecx
1f34:   lea     esp,[esp-0x4]
1f38:   popf
1f39:   ret
...
```

a) disassembled code (comment manually added)

```c
int check(char *param_1) {
  return (int)*param_1;
}
```

b) decompiled code

c) control flow graph

Figure 7: Disassembled / decompiled view of obfuscated program with ROPonly

```
1eec:   mov     ecx,0x41
1ef1:   xchg    edx,eax
1ef2:   mov     eax,edx
1ef4:   sub     eax,ecx
1ef6:   xchg    edx,eax
1ef7:   jmp     1f0d


1f0d:   je      1f5e
1f13:   jmp     1f3a
```

a) disassembled

```c
undefined8 check(char *param_1) {
  int iVar1;
  undefined4 local_4;

  iVar1 = (int)*param_1 + -0x41;
  if (iVar1 == 0) {
    iVar1 = (int)param_1[1] + -0x42;
    if (iVar1 == 0) {
      iVar1 = (int)param_1[2] + -0x43;
      if (iVar1 == 0) {
        iVar1 = (int)param_1[3];
        if (iVar1 == 0) {
          local_4 = 1;
        } else {
          local_4 = 0;
        }
      } else {
        local_4 = 0;
      }
    } else {
      local_4 = 0;
    }
  } else {
    local_4 = 0;
  }
  return CONCAT44(iVar1,local_4);
}
```
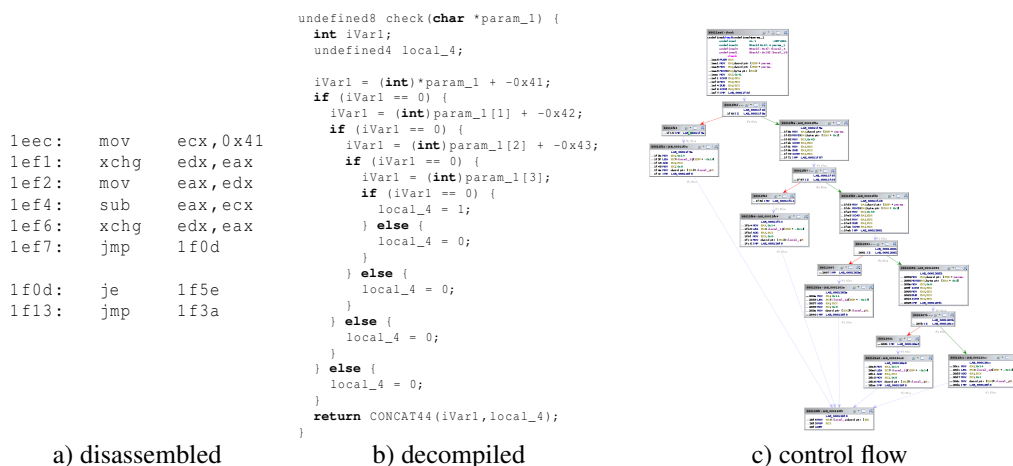
b) decompiled

c) control flow

Figure 8: Disassembled / decompiled view of the program (ROPonly), deobfuscated by static ROP analyzer

```
push    eax
mov     eax,dword ptr [esp+0x8]
mov     eax,dword ptr [esp+0x8]
movsx   eax,byte ptr [eax]
lea     esp,[esp-0x1c]
push    eax
push    ecx
push    edx
lea     esp,[esp+0x28]
mov     eax,0x6281e212
mov     edx,0x3d213e44
mul     edx
cmp     eax,0xe4c7afa1
setne   al
cmp     edx,0xb7dc5433
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
mov     eax,0x517f01b6
mov     edx,0x69497d86
mul     edx
cmp     eax,0x7dc83b19
setne   al
cmp     edx,0xd81f4da3
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
mov     eax,0x57d01e6a
mov     edx,0x207cee30
mul     edx
cmp     eax,0x9f86c923
setne   al
cmp     edx,0xf36a0983
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
mov     eax,0x69f9a1f9
mov     edx,0x3f918ca
mul     edx
cmp     eax,0x9508b8df
setne   al
cmp     edx,0xb932e736
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
...
    (snip)
...
mov     eax,0x150bdf2a
mov     edx,0x3e684931
mul     edx
cmp     eax,0xe0cceb4d
setne   al
cmp     edx,0x88c417ac
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
mov     eax,0x309d4ff9
mov     edx,0x27ed7d5e
mul     edx
cmp     eax,0xb88f7e35
sete    al
cmp     edx,0xbccb864d
sete    dl
and     al,dl
shl     ecx,1
or      cl,al
mov     eax,0x58c1bacc
mov     edx,0x16ced49a
mul     edx
cmp     eax,0x548a51a1
setne   al
cmp     edx,0xbadf6397
setne   dl
or      al,dl
shl     ecx,1
or      cl,al
mov     eax,ecx
xor     eax,0x48272b13
add     eax,fts_children
push    eax
lea     esp,[esp-0xc]
pop     edx
pop     ecx
pop     eax
ret
```

a) disassemble code

```
undefined8 check(undefined4 param_1,
                 undefined4 param_2,
                 char *param_1_00) {
  return CONCAT44(param_2,(int)*param_1_00);
}
```

b) decompiled code   c) control flow

Figure 9: Disassembled / decompiled view of obfuscated program with ROP+OP_Basic

```
mov     eax,0x4f772ed5
mov     edx,0x911add0b
mul     edx
xchg    DWORD PTR [esp-0x10],eax # hidden code
xchg    DWORD PTR [esp-0x1c],eax # hidden code
xchg    DWORD PTR [esp-0x10],eax # hidden code
cmp     eax,0xd96363cb
sete    al
cmp     edx,0x73167dc0
sete    dl
and     al,dl
shl     ecx,1
or      cl,al
mov     eax,0x31831fd0
mov     edx,0x179daa15
mul     edx
add     DWORD PTR [esp-0x20],0x749f1b7f # dummy code
cmp     eax,0xca2ccea1
sete    al
cmp     edx,0xf34b29e6
sete    dl
...
setne   dl
or      al,dl
mov     eax,ecx
xor     eax,0x7731540a
add     eax,gnu_get_libc_release
push    eax
```

Figure 10: Disassembled code of program obfuscated with ROP+OP_DSE+Hiding