# ROPfuscator: Robust Obfuscation with ROP

Giulio De Pasquale*†, Fukutomo Nakanishi*‡, Daniele Ferla§, Lorenzo Cavallaro¶
†King's College London ‡Toshiba Corporation §Università di Bologna ¶University College London

***Abstract*—Software obfuscation is crucial 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) became one of the most effective exploitation techniques for memory error vulnerabilities. ROP interferes with our natural perception of a process control flow, inspiring 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 present ROPFuscator, a compiler-driven obfuscation pass based on ROP for any programming language supported by LLVM. 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. The project's source code is published online to aid further research.**

## I. 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. As such, software is widely recognized as an important intellectual property to protect from reverse engineering attempts [1], [2], [3].

In this context, obfuscation represents the de-facto standard for protecting software from being disclosed, making reverse engineering prohibitively expensive, i.e., allowing a vendor to collect enough revenue at the peak of a marketing campaign. 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], anti-debugging [8] and mixed-boolean arithmetic (MBA) [9], [10], [11] are well-known software obfuscation techniques. They all exploit assumptions that challenge—one way or another—the logic of reverse engineering algorithms. For instance, inserting dummy code and opaque predicates interferes with attempting to reconstruct an underlying algorithm's semantics directly. In contrast, control flow flattening breaks the ability to understand a program's execution flow, challenging 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 techniques [12]. Core to this is the ability to chain the invocation of chunks of code (gadgets) to execute arbitrary - and often malicious - code. As such, ROP builds its working logic on threaded code [13], changing the usual interpretation 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 of the obfuscation's granularity, operating at the level of individual assembly instructions. Third, an obfuscated piece of code would see its semantics due to the execution of code gadgets scattered throughout the entire process address space.

Prior work already explores the effectiveness of ROP as an obfuscation technique. For instance, Mohan et al. [14], and Borrello et al. [15] repurpose ROP to challenge malware detection tasks. Teuwen et al. [16] introduced the first patent on ROP as an obfuscation technique; Mu et al. [17] 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 to explore 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 [18], where reverse engineering attempts tailored at ROP-based obfuscation [19], [20], [21] or dynamic symbolic execution [22] still undermine its effectiveness. In another work, Borrello et al. [23] proposed a binary-rewriting approach that uses ROP chains coupled with a custom algorithm to protect its gadget addresses.

We present ROPFuscator, a compiler-driven obfuscation pass for any programming language supported by LLVM. At its core, ROPFuscator relies on ROP microgadgets [24] to obfuscate arbitrary programs at the granularity of individual assembly instructions. To withstand attacks of increasing sophistication, ROPFuscator relies on opaque predicates and constants [25] to protect the recovery of the gadget locations in the address space. We present a thorough reproducible evaluation of ROPFuscator across five dimensions, which supports our principled reasoning with evidence on completeness (code coverage), incurred overhead, correctness, robustness to MATE attacks, and practicality. We release ROPFuscator to support further the need for principled reasoning in domains characterized by an 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 (§II). This helps us to provide principled reasoning to identify and justify the design choices that avoid brittle arms races that are endemic to the software obfuscation domain instead of providing

* Equal contribution

researchers and practitioners with a clear and contextualized understanding of strengths and limitations.
- We present ROPFuscator, a compiler-driven obfuscation pass based on ROP (§III-B) for any programming language supported by LLVM. To withstand sophisticated MATE attacks, we equip ROPFuscator with opaque predicates and constants (§III-C), 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 (§III-C0c).
- We present a thorough evaluation of ROPFuscator along five dimensions to support our principled reasoning and provide the opportunity to understand its effectiveness in practical contexts (§IV).

## II. Threat Model

Our threat model considers MATE attacks of increasing sophistication [18]. In particular, we assume attackers can rely on static ROP-agnostic and ROP chain disassembly analyses, dynamic symbolic execution, and ROP-specific dynamic analyses. In doing so, we adopt metrics similar to the one used in previous work on software obfuscation [3], [26].

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.

*a) Threat A: ROP-agnostic Static Analysis:* The core of 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.

*b) Threat B: Static ROP Chain Analysis:* It is perhaps unsurprising that ROP chains' introduction in a program naturally breaks traditional disassembly algorithms. They assume a code execution model relying on an architecture-specific instruction pointer register; on the other hand, ROP, built on threaded code, reuses the stack pointer register - or, more generally, any other general-purpose register - to keep track of the next instruction to execute. A more realistic threat model here should thus consider the ability of statically analyzing ROP chains to reconstruct the original control flow of the obfuscated program, which can be leveraged to build more insightful data flow and decompilation analyses.

*c) Threat C: Dynamic Symbolic Execution:* Static ROP chain analysis requires identifying 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 following 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.

*d) Threat D: Dynamic ROP Chain Analysis:* This analysis takes advantage of runtime information in a context where 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 [20], [27].

## III. Architecture and Implementation

In the following sections, we present a more detailed view of ROPFuscator, show its obfuscation steps, and how they are interconnected.

### A. Architectural Overview

Our framework obfuscates LLVM-produced code at the x86 assembly level. The source code is compiled into LLVM's IR and processed by ROPfuscator. It consists of three components called in subsequent order, shown in Figure 1, named ROP transformation, opaque predicate insertion, and instruction hiding.

The first, *ROP transformation* (§III-B) converts instructions into ROP chains, while the second one *Opaque predicate insertion* (§III-C) injects opaque predicates in the ROP chain generation code to protect the gadgets' entry point address and other program immediates. Finally, *instruction hiding* (§III-C0c) picks some instructions and embeds 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 should be generally used after executing the ROP transformation.

### B. ROP transformation

Methods of converting normal code to equivalent gadgets are proposed in several studies [28], [14]. However, instead of processing native machine instructions, they transform various intermediate representations into ROP gadgets. In our work, we take a significantly different approach. As shown in Figure 2, ROPFuscator decomposes a native x86 instruction into multiple instructions, which are then matched with the available microgadgets. The address of the gadgets, along with the symbol table of the target library, is used to emit the obfuscated ROP code. We explain this process in detail in the following paragraphs.

*a) Gadgets extraction:* The extraction process is based on the Galileo algorithm [12]. The algorithm identifies `ret` instructions and scans in reverse to locate a valid instruction sequence, the ROP gadget. The gadgets are extracted from a shared library chosen by the user. For design simplicity, we only rely on *microgadgets* [24] of length 1 (i.e, only one instruction before the `ret` instruction) to build ROP chains.

*b) ROP chain generation:* The use of microgadgets may incur 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 2). The temporary registers are found by performing live register analysis [29] 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. [24], to generate ROP chains (Step (ii) in Figure 2).

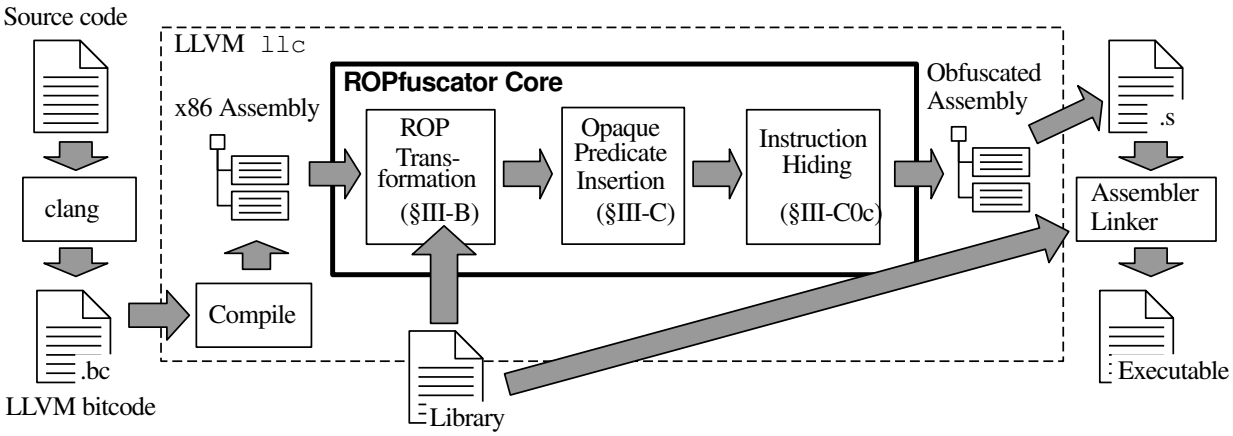


Fig. 1: An architectural view of ROPFuscator.

*c) Emitting ROP generator code:* Once the gadgets are extracted, the ROP chain must 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 2). A gadget address is calculated using the address of a random symbol from the linked library as a base address to deal with ASLR. 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.

*d) Program semantics preservation:* It is crucial to preserve the semantics of the program while obfuscating. One of the peculiarities of the x86 ISA is that it contains many instructions that can modify the flag register. Therefore, we carefully use the `pushf` and `popf` instructions to save and restore the flags on necessity. Fortunately, since LLVM exposes valuable metadata to check whether register flags may be safely clobbered, we can use this information to omit flag saving when unnecessary and thus drastically improve the performance.

### C. Opaque Predicates and Instruction Hiding

This section will discuss how opaque predicates are generated and later inserted into 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 [30] and computationally hard programs (e.g. pointer-aliasing [6], 3SAT [25], [31]).

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 extensive and is used as a basis of the RSA cryptosystem [32]. The second algorithm, *Random 3-SAT*, is based on Sheridan et al. [31]. 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 previous results, which suggest $N \geq 6$ to have high chances for the clauses to be unsatisfiable. The formula is then negated, forcing the output always to be 1.

Based on internal results, opaque predicates that used random 3-SAT yielded a worse size-to-performance ratio and were less robust against DSE. We evaluated our approach exclusively using the integer factorization algorithm for this reason.

*a) Opaque Gadget Addresses Against Static Analysis:* We use opaque predicates to protect gadget addresses and immediate operands (Step (iv) in Figure 2). An opaque predicate can generate a 1-bit output that is hard to compute statically.

For this reason, we concatenate each output bit of 32 distinct opaque predicate instances to generate a 32-bit constant (*opaque constant*). We chose the arbitrary length of 32-bit for the constants since our work focused on a 32-bit architecture: naturally, this approach can be easily extended to different 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.

*b) DSE-resistant Opaque Predicates:* In the previous section, we discussed the generation of opaque predicates and their application to protect the gadget addresses against static analysis. However, this is not robust enough against DSE attacks (threat C). This section focuses on the steps we undertook to make the predicates DSE-resistant. We evaluated our approach with angr [22], 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, resulting in 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 unknown and concretized, the symbolic execution engine needs to evaluate the opaque predicates symbolically. This will force the computation of the mathematically complex problem on which the opaque predicate is based. Thus, we focused on finding appropriate input that cannot be easily concretized, imposing additional calculations on the symbolic execution engine.

*c) Instruction Hiding:* Instruction hiding consists of the decomposition of an instruction into suboperations, of which, some are selected to be inserted out-of-order in other neighboring locations, protected by opaque predicates and the addition of

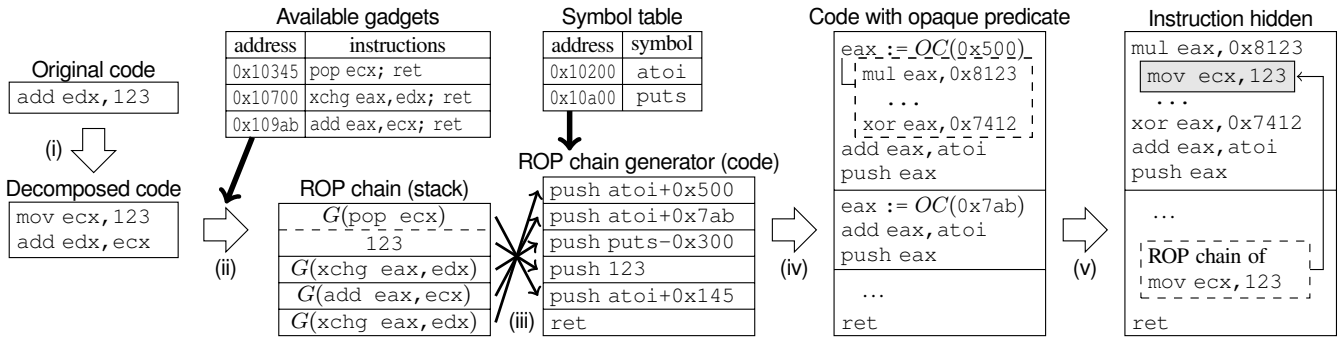


Fig. 2: Obfuscation Transformation Example with ROP, Opaque Predicates, and Instruction Hiding.

dummy code. The reordering is applied exclusively to instructions whose order does not affect the outcome of the calculation. We describe the process in more detail as follows.

*d) Hidden Code Selection:* In this step, the pass decides which part of the code is to be obfuscated by instruction hiding or by ROP transformation. It is essential to balance the two 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 instructions 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.

*e) 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 §III-B.

*f) 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. Besides useless computing operations, the code modifies the predicate's internal state and its variables, leading to confusion for the attackers.

## IV. Evaluation

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

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

We measure the obfuscated instructions' coverage *(RQ1)* and performance overhead *(RQ2)* in §IV-A, and §IV-B respectively. During the test runs, we observe whether the program is executed correctly *(RQ3)* and we evaluate our approach robustness *(RQ4)* against the attacks defined in §II in §IV-C. Finally, we discuss practicality *(RQ5)* by applying ROPFuscator onto an open-source media player.

We evaluate several obfuscation configurations throughout this section. In particular, we select the following configurations that showcase varying degrees of obfuscation:

- *Baseline*: non obfuscated binaries
- *ROPonly*: ROP transformation only
- *ROP+OP*$_{\text{Basic}}$: ROP transformation, along with basic opaque predicates
- *ROP+OP*$_{\text{DSE}}$: ROP transformation, along with DSE-resistant opaque predicates
- *ROP+OP*$_{\text{DSE}}$*+Hiding*: ROP transformation with opaque predicates and instruction hiding

*a) Test Sets and Experiment Environment:* We evaluate coverage, execution speed (throughput), and code size with two test sets, the *SPEC CPU2017* (SPECrate Integer) C/C++ tests and a subset of *binutils*' applications. We evaluated the robustness with a simple program that validates a user-provided buffer. Unless explicitly stated, the test cases are compiled with no optimizations (`-O0` flag), the gadgets are extracted from the 32-bit libc version 2.27-3ubuntu1, and the programs are executed in a virtual machine running Ubuntu 18.04 x86-64.

### A. Completeness: Obfuscation Coverage

We first evaluate the ratio of obfuscated instructions when the sole ROP transformation pass is enabled. ROPFuscator is more robust when instructions are transformed since static disassemblers do not reconstruct the code executed by ROP gadgets.

Table I presents the ratio of obfuscated instructions in SPEC CPU test cases with optimization options `-O0` and `-O3`. When compiler optimizations are disabled (`-O0`), about 60–80% of the instructions are obfuscated into ROP chains. However, when all optimizations are enabled (`-O3`), the average ratio decreases to around 40%. This is because x86 is a CISC architecture, which makes it difficult to translate multiple specialized instructions into ROP microgadgets. Therefore, avoiding compiler optimizations may be more beneficial.

About 7–12% of the instructions were not obfuscated due to the absence of accessible registers or gadgets, noting that the optimization levels did not impact this metric. It is possible to decrease this gap by saving and restoring registers to allocate free temporary registers or linking an ad-hoc library to access more gadgets.

Next, we observe the obfuscation coverage when linking our test cases against different library versions. Table II shows the ratio of instructions obfuscated for two programs in *binutils 2.32*, while Table III focuses on the performance overhead.

TABLE I: 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% |

TABLE II: 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 | Obfuscated | 82.69% | 80.93% |
| | Unobfuscated (No gadget / reg) | 6.35% | 5.77% |
| | Unobfuscated (Other) | 10.96% | 13.31% |

TABLE III: Runtime slowdown and code size of obfuscated programs for binutils for each obfuscation algorithm.

| | | absolute value | | ratio (Baseline=1) | | ratio (Roponly=1) | |
|---|---|---|---|---|---|---|---|
| metric | obfuscation | 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$_{\mathrm{Basic}}$ | 41.1s | 337s | 107 | 1118 | 9.7 | 11.0 |
| | ROP+OP$_{\mathrm{DSE}}$ | 66.4s | 761s | 172 | 2527 | 15.7 | 24.8 |
| | ROP+OP$_{\mathrm{DSE}}$+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$_{\mathrm{Basic}}$ | 895MB | 1407MB | 828 | 1269 | 86.6 | 89.7 |
| | ROP+OP$_{\mathrm{DSE}}$ | 1530MB | 2411MB | 1417 | 2175 | 148 | 154 |
| | ROP+OP$_{\mathrm{DSE}}$+Hiding | 1283MB | 2063MB | 1188 | 1861 | 124 | 132 |

There is a coverage decrease, around 20–40%, between libc versions *2.27-3ubuntu1* and *2.27-3ubuntu1.2*. Furthermore, the instructions that are not obfuscated due to missing gadgets increase by 50–60% on average. This is due to libc *2.27-3ubuntu1.2* missing a gadget[1] widely used by ROPFuscator. As previously described, we rely on exchange gadgets to convert single instructions into a combination of microgadgets [24]. In this case, if a similar gadget is unavailable, it is impossible to use microgadgets relying on registers that must be exchanged first. Therefore, ROP transformation is very sensitive to ROP gadgets' availability, per the results obtained when linking a different libc version (*2.31-0ubuntu9*). A straightforward solution to this problem would be to find such a gadget elsewhere, for instance, by linking a library that contains this gadget to the target application.

**Takeaway - RQ1: Completeness**.

On average, ROPFuscator transforms about 60–80% of the instructions into ROP chains. The number depends on the compiler optimization option and shows better coverage without optimization. The number also depends mainly on the library version from which the ROP gadgets are extracted, and selecting an appropriate library version ensures high coverage.

### B. Performance and Correctness

This subsection focuses on the run-time slowdown and code size overhead introduced by ROPFuscator.

In Figure 3, ROP transformation causes a 140x (109-189x) mean execution time increase and a 13x (10-16x) binary size increase. Assessing opaque predicate obfuscation passes was unfeasible for SPEC CPU tests, so performance and code size overhead were evaluated on a binutils' applications subset.

Table III reveals that opaque predicate obfuscation raises execution time by 10x without DSE resistance (*ROP+OP*$_{\mathrm{Basic}}$) or 20x with DSE countermeasures (*ROP+OP*$_{\mathrm{DSE}}$). Code size increases by roughly 90x for basic opaque predicates and 150x for DSE-resistant ones. Instruction hiding (*ROP+OP*$_{\mathrm{DSE}}$*+Hiding*) reduces ROP gadgets, slightly improving performance with a 16x longer execution time and a 130x larger size.

Taking *Baseline* as a reference, the execution time is 10–200x longer with ROP transformation only, 200–4000x with ROP alongside DSE-resistant opaque predicates, and 150–3000x with all the obfuscation passes enabled. Simultaneously, the executable sizes are 10–16x bigger with ROP transformation only, 1500–2500x with ROP and DSE-resistant opaque predicates, and 1200–2000x with full obfuscation.

The overhead ratio is undoubtedly significant when obfuscating an entire program; however, heavy obfuscations are generally focused on specific portions of a program [23]. In the following §IV-D we show how limiting the obfuscation does not impact the usability of real-world software.

Finally, we observed no behavioral differences from the original programs during the experiments. We compared the obfuscated programs' output and checked whether it matched one of the unobfuscated versions. Though it is not formally proven to be correct, ROPFuscator could accurately preserve the program semantics in our experiments.

**Takeaway - RQ2-RQ3: Performance and Correctness**.

ROP transformation imposes 10–200x execution time and 10–15x code size overhead. DSE-resistant opaque predicates impose an additional 20x increase (total 200–4000x) in the execution time and a further 150x (total 1500–2500x) code size increment. While the results show significant overheads, controlling the performance/size overhead ratio is possible by carefully choosing critical locations that may be strongly obfuscated. A summary and robustness evaluation are shown in Table IV. Finally, ROPFuscator preserved the semantics of the tested programs throughout the obfuscation transformations.

### C. Robustness

This section will refer to two programs whose source code is listed in Figure 4. These examples represent simple input code validation routines that must be protected from the analysis. We propose two versions of the source code, early and late exit, due

[1] `xchg eax, edx; ret`

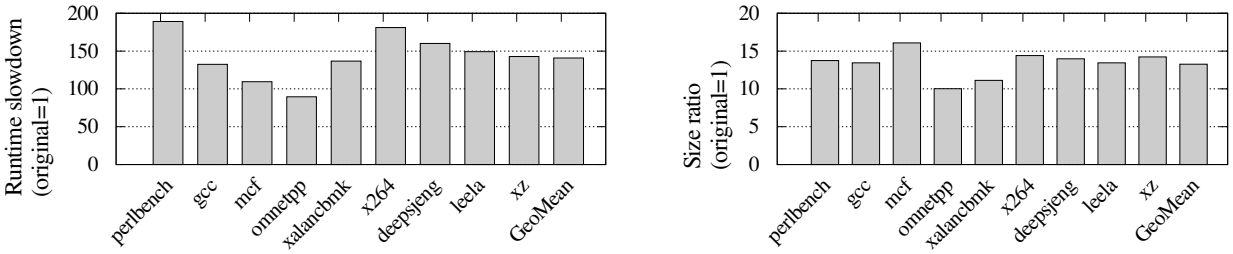


Fig. 3: Runtime slowdown and code size of obfuscated programs for SPEC CPU 2017 (SPECrate Integer).

to the different exploration behavior by symbolic execution engines when evaluating a return instruction. We evaluate robustness taking into account the threat categories highlighted in the threat model defined in §II: threats A *(ROP-agnostic Static Analysis)*, B *(Static ROP Chain Analysis)*, C *(Dynamic Symbolic Execution)* and D *(Dynamic ROP Chain Analysis)*. We used open-source decompilers to reverse-engineer the binaries: Ghidra, retdec, and r2dec. The results of this evaluation are summarized in Table IV.

(a) early-exit function

```
1 int check(const char *s) {
2
3   if (s[0] != 'H') return 0;
4   if (s[1] != 'e') return 0;
5   if (s[2] != 'l') return 0;
6   ...
7   if (s[12] != '\0') return 0;
8   return 1;
9 }
```

(b) late-exit function

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

```
10
11        int main(int argc, char **argv) {
12          if (check(argv[1])) puts("OK");
13            return 0;
14        }
```

Fig. 4: Example of an input validating program.

*a) Threat A: ROP-agnostic Static Analysis:* We evaluate robustness against this threat by decompiling the functions listed in Figure 4. The code is compiled with and without obfuscations and processed by reverse engineering tools.

The results show that all decompilers can reconstruct the functions that have not been obfuscated successfully but fail in the presence of the ROP transformation pass. This is likely due to the decompilers' inability to correctly dissect the ROP chains' structure and recognize the function boundaries. The opaque predicates are decompiled successfully but require extensive human analysis to be removed from the decompiled code.

*b) Threat B: Static ROP Chain Analysis:* We implemented a deobfuscator to dissect and extract the ROP chains to the original code, using a similar technique to deRop [19]. We first identify the gadgets and later combine their underlying code into the original instructions. We do not process the ROP chains directly in memory; instead, our deobfuscation approach statically detects the ROP chain generator code by disassembling the binary, looking for stack manipulating instructions such as `push`, `pop` and `ret`. Once the ROP chain is detected, we emulate the stack contents' execution, extracting the gadgets and concatenating their underlying code. We deobfuscated binaries protected with the ROPonly and ROP+$OP_{DSE}$ configurations.

The recovered instructions from the *ROPonly* binary are slightly different from the original code, and the decompiled code has strong similarities and structure to the original C source code. On the other hand, our deobfuscator could not recover the code without fully emulating the opaque predicates' execution in the *ROP+$OP_{DSE}$* binaries. Employing a CPU emulator diverges from this threat context as it would be more appropriate to Threat D, Dynamic ROP Analysis. Therefore, we believe that a targeted static ROP chain analysis can tackle the ROP transformation but cannot handle the opaque predicates.

Besides, we analyzed the immediate operands that appear in the ROP chains. In this case, we simulate a scenario where an attacker knows the application's constants. The example shown in Figure 4 is reflected by the comparisons in lines 3–7. Those statements are converted into a recognizable assembly pattern [2] in the ROP chain, facilitating the extraction of the string constant. For this reason, we focused on extracting data from such patterns (i.e., periodic occurrence of immediate operands with various intervals). We successfully recovered the string constants in ROPonly binaries but failed to do so from ROP+$OP_{DSE}$ ones. This shows the effectiveness of protecting immediate operands with opaque predicates.

*c) Threat C: Dynamic Symbolic Execution:* For this scenario, we considered an automated attack for computing an input able to pass validation checks. We use the *angr* symbolic execution engine to find such input that causes the application shown in Figure 4 to output '`OK`.' We obfuscated the program with every obfuscation configuration.

We employed different symbolic exploration strategies: depth-first vs. breadth-first search and symbolic vs. tracing. Additionally, we set a memory use threshold of 8GB and measured the time and memory needed to compute the input. The results are shown in Table V.

As expected, DSE is very effective in analyzing Baseline, ROPonly (less than 10 seconds) and ROP+$OP_{Basic}$ binaries (less than 2 minutes). On the other hand, it fails to find a valid input for ROP+$OP_{DSE}$ binaries due to the exploration's significant memory requirements. Combined with DSE-resistant opaque predicates, ROP severely hinders input-finding attacks with symbolic execution techniques, even for a straightforward program like the one used in our tests.

*User input in opaque predicates.* We observed that the sole application of a ROP transformation or opaque predicates independent from user input is not enough to protect against DSE. As any branch based on user input values increases the time complexity

[2] e.g., `push 0x48; push` *G*(`pop ecx`); `...`

TABLE IV: Robustness and performance of each algorithm in ROPFuscator against attacks.

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

○: Breakable, ●: Robust, ◕: Mostly Robust

of DSE, we compute the gadget addresses using opaque constants intertwined with such input. We believe our approach should achieve similar results to the ones presented by Banescu et al. [33]

Additionally, we propose a hypothesis that would challenge DSE: frequently used general-purpose registers, such as `eax` on the x86 architecture, might be used to handle user input. We empirically verified our assumption by disassembling the compiled example code and noticed that each input byte is processed into the `eax` register before being compared against an expected value.

This result indicates that our obfuscation measures introduced in III-C, such as feeding user-supplied registers as input to opaque predicates, are effective against DSE.

*d) Threat D: Dynamic ROP Chain Analysis:* Finally, we evaluate the robustness of ROPFuscator against Dynamic ROP Chain Analysis. As previously noted, static analysis cannot efficiently analyze opaque predicates. For this reason, we emulate the ROP chain building code and subsequently implement a dynamic deobfuscator, adopting a similar approach to those proposed in previous works [20], [27]. The deobfuscator analyzes an execution trace generated by a CPU simulator for possible ROP chains. Furthermore, the trace is extracted from a code region provided by the user.

We applied the dynamic ROP deobfuscator to the early-exit function shown in Figure 4 (a) compiled with the configurations ROP+OP$_{\mathrm{DSE}}$ and ROP+OP$_{\mathrm{DSE}}$+Hiding. We successfully extracted a semantically equivalent version of the code from the ROP+OP$_{\mathrm{DSE}}$ obfuscated binary. On the other hand, the code extracted from the binary obfuscated with ROP+OP$_{\mathrm{DSE}}$+Hiding was only *partially* matching the original one. This is due to instructions being hidden in the opaque predicates' body and, therefore, being ignored by the deobfuscator during analysis.

As a result, the sole use of opaque predicates is not robust against the dynamic tracing attack. Conversely, instruction hiding appears effective when combined with predicates, preventing code from being revealed in an execution trace.

**Takeaway - RQ4: Robustness**.

ROP transformation (ROPonly) is robust against Threat A but not robust against Threat B, C, and D. Introducing opaque predicates (ROP+OP$_{\mathrm{DSE}}$) fortifies the programs against Threats B and C. Finally, instruction hiding (ROP+OP$_{\mathrm{DSE}}$+Hiding) makes the obfuscated binaries resistant against Threat D. The results are summarized in Table IV.

### D. Practicality: Case Study

In §IV-B, 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, are subject to obfuscation interest take up a small portion of the total execution time. To balance robustness and performance, we considered obfuscating such functions selectively, and we later verified our assumptions in the rest of this section.

We apply ROPFuscator to an open-source media player, VLC. We explain how obfuscation mechanisms can be applied to protect critical assets in the program, balancing robustness and performance.

Digital rights management (DRM) is a mechanism to protect commercial media content against digital piracy [34]. It is extensively used to protect video, audio, video games, and other media distributed on the Internet from unauthorized use.

The foundation of DRM is the encryption (or scrambling) mechanism to protect content. Attackers are interested in retrieving the encryption keys to decrypt the content to bypass the protection and illegally use or copy the material. Obfuscation plays an essential role in dissuading reverse engineering attempts that may disclose the algorithms and encryption keys employed by the DRM protections.

*a) Protecting the DRM encryption routines:* We applied ROPFuscator to an open-source media player, VLC Media Player, using the DVD descrambling library `libdvdcss`[3]. Upon VLC Media Player's request, `libdvdcss` library derives the content decryption key (title key) from the protected DVD media and decrypts (descramble) the DVD content, which is later decoded and played by the media player on screen.

Though the final goal is to protect media content, we believe it is crucial to protect the title keys and the key derivation process. Therefore, we prioritized the obfuscation of the title key derivation more than the content decryption functions with the *Balanced* configuration, ultimately aiming to achieve a better performance/size overhead ratio.

*b) Evaluating the obfuscation impact:* We obfuscated `libdvdcss` with five configurations and then compared their performances:

- *Baseline*, no alterations made to the code
- *ROPonly*, *all* functions obfuscated with ROP transformation only
- *ROP+OP*$_{\mathrm{DSE}}$, *all* functions obfuscated with ROP transformation and DSE-resistant opaque predicates
- *ROP+OP*$_{\mathrm{DSE}}$*+Hiding*, *all* functions obfuscated with ROP transformation, DSE-resistant opaque predicates, and instruction hiding
- *Balanced*, *only title key derivation* functions obfuscated with ROP+OP$_{\mathrm{DSE}}$+Hiding and the rest of the library with ROPonly

[3]https://www.videolan.org/developers/libdvdcss.html

TABLE V: DSE performance overhead 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+$OP_{\mathrm{Basic}}$ | 85.3s | 417MB | 57.4s | 365MB | 85.7s | 413MB | 56.1s | 368MB |
| | ROP+$OP_{\mathrm{DSE}}$ | Out of Memory | | Out of Memory | | Out of Memory | | Out of Memory | |
| | ROP+$OP_{\mathrm{DSE}}$+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+$OP_{\mathrm{Basic}}$ | 69.7s | 324MB | 68.2s | 326MB | 74.1s | 342MB | 74.0s | 345MB |
| | ROP+$OP_{\mathrm{DSE}}$ | Out of Memory | | Out of Memory | | Out of Memory | | Out of Memory | |
| | ROP+$OP_{\mathrm{DSE}}$+Hiding | Out of Memory | | Out of Memory | | Out of Memory | | Out of Memory | |

Out of Memory: force stopped after exceeding 8000MB

TABLE VI: Performance statistics of VLC Media Player using `libdvdcss`.

| 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+$OP_{\mathrm{DSE}}$ | 110.2 | 97.0% | No | 48.5 |
| ROP+$OP_{\mathrm{DSE}}$+Hiding | 120.7 | 95.3% | No | 41.3 |
| Balanced | 30.2 | 23.2% | Yes | 18.4 |

We tested each of these configurations by playing a commercial DVD title for 30 seconds[4], using VLC Media Player linked with `libdvdcss`. The playback results are listed in Table VI.

We measured the number of function calls and instructions using Valgrind [35]. `libdvdcss` accounts for about 5.5% of the total instructions executed. Among `libdvdcss`, the main DVD decryption function, `dvdcss_unscramble`, accounts for 99.88% of the instructions (15091 out of 15970 function calls), while the key derivation functions only for 0.028% of the instructions (63 function calls). Therefore, using a slower albeit more robust obfuscation for 0.0015% of the overall instructions (0.028% of `libdvdcss`) does not significantly impact performance.

The average CPU usage is calculated by dividing the CPU utilization time by the total execution time[5]. The results show that, without obfuscation, the program plays the DVD video smoothly, with the CPU usage averaging around 10%. The application still plays the video smoothly when the sole ROP transformation is applied, albeit with an additional CPU 10% usage on average. Applying opaque predicates upon the ROP transformation renders the media not playable in real-time, as the player frequently stops to buffer the movie contents and to peak the CPU usage to almost 100%. Finally, using the *Balanced* profile shows that the obfuscation overhead is almost on par with the sole ROP transformation while retaining the real-time decoding of the media.

These results seem to support our assumptions: retaining a reasonable performance/size ratio when obfuscating applications by targeting sensitive functions is possible. In this case, we could fortify the encryption routines while retaining the intended user experience.

## V. 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 §IV-A).

Our experiments highlighted the trade-off between the performance and robustness of our approach (Table IV). However, tuning the obfuscation layers can be balanced per function (§IV-D).

Although ROPFuscator is not specifically designed for data obfuscation, it can protect constant values by leveraging other techniques to work synergically with our approach. In the current implementation, we chose opaque constants to protect immediates in the binaries generated by ROPFuscator. Moreover, users can embed and protect sensitive data such as constants used in white box cryptography [36] and recent developments in Mixed-Boolean Arithmetic (MBA) obfuscation [9] suggest other promising ways to strengthen ROP obfuscation.

*a) Portability of the obfuscated program:* Extracting gadgets from commonly used libraries, seemingly the natural path to follow, makes the obfuscated binary locked on the specific library chosen at compile time and, subsequently, not portable. For this reason, it is recommended to choose a library that ships with the program itself or create one from the ground up. In lieu of future work, we developed an accessory library called LIBROP[6] providing user-specified gadgets and removing the limitations of using system libraries.

Another solution to having a fully portable program is to link the library used to extract the gadgets statically.

[4] `time vlc -I dummy dvd://#1:3 --stop-time=745 vlc://quit`

[5] $(T_{\mathrm{user}}+T_{\mathrm{system}})/T_{\mathrm{real}}$

[6] https://github.com/ropfuscator/librop

*b) On the use of microgadgets:* We adopted microgadgets for design simplicity and to speed up the prototyping cycle(III-B). While microgadgets are easier to extract, mapping a single instruction to one or more microgadgets skews the performance of the obfuscator. Spatial and time overhead depends on the number of instructions of a gadget and the number of gadgets needed to replace a specific instruction. The more gadgets are needed to replace a certain instruction, the higher the overhead. This characteristic aspect of ROP is an interesting research direction and should be pursued further as it could significantly impact the technique's overall performance.

*c) On the support of additional architectures:* ROPfuscator is designed for the 32-bit x86 instruction set as a prototype. Although supporting other architectures is achievable with further development, the core research concept remains unaffected by the present implementation.

*d) Practicality:* ROPfuscator can be employed as a *selective* obfuscation technique for obfuscating specific sensitive functions. However, its application to the entire code base is impractical due to the severe performance restrictions. Consequently, we advocate for exploring alternative directions while improving ROPfuscator key components' implementation, such as researching the impact of variable-length gadgets.

*e) Resistance to other deobfuscation approaches:* First, we implemented a countermeasure to opaque predicate identification attacks [37], [38]. We could not test the previous works for different reasons. We encountered technical issues in compiling the project by Ming et al. [37], that promptly offered us support. Unfortunately, the issues persisted. Differently, we could not evaluate against the work by Tofighi-Shirazi et al. [38] and Yadegari et al. [21] due to the code being unavailable or not functioning at the time of writing.

We also examined VMHunt [39], a virtualization deobfuscation method that claims compatibility with ROP chains. Regrettably, it was unable to identify any ROP gadgets in our experiments.

Furthermore, we tested Syntia, a program synthesis-based deobfuscation framework. It proved effective with simple functions but faced difficulties with more complex ones, irrespective of obfuscation, making it less suitable for our approach.

Finally, we considered a scenario where an attacker has complete knowledge of ROPFuscator and its mechanics. Although it is impossible to completely prevent an attacker from tracing ROP chain execution, instruction hiding offers some protection by disrupting the execution trace. This aligns with our objective of increasing analysis time cost.

## VI. Related Work

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

*a) ROP-based obfuscation:* ROP is applied in various ways to protect software. Recently, Borrello et al. [23] proposed an obfuscation approach based on ROP chains protected with a custom algorithm. This approach is based on static binary rewriting, i.e., a set of techniques to modify existing executable programs without the aid of a compiler and access to source code. As such, the developer must overcome challenges not present when hooking into a compiler, for instance, computing register liveness or finding a function's boundaries. Binary approaches are generally more error-prone and less resilient than techniques based on the compiler's visibility over the object files [40]. However, we acknowledge that these approaches are more generic as they can be applied to closed-source software.

Another common application is to evade detection from signature-based software such as anti-virus solutions. Frankenstein [14] extracts ROP gadgets from benign binaries and combines them to generate ROP chains that execute malicious actions. However, this approach is not robust enough to counter an attack in a MATE scenario since it is not designed to withstand targeted ROP analyses. ROP needle [15] 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 [41] proposes a mechanism to embed ROP gadgets into sensitive code regions. Modifying these code regions leads to the ROP chain being corrupted, impeding its proper execution. Since they inherently change the program code, this can deceive debuggers when setting software breakpoints. 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 [42], [17]. RopSteg [42] 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), excluding a targeted ROP chain analysis executed through dynamic tracing (Threat D). ROPOB [17] 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).

*b) ROP Chain Generation:* Q [28] 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 into a ROP chain. 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.

*c) 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 [30], and computationally hard problems (e.g., pointer-aliasing [6] or 3SAT [25], [31]). 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 [43]. *Dynamic* opaque predicates introduce the idea of correlated predicates that map a predicate's output to the input of a subsequent one [44].

These proposals are orthogonal to our approach, i.e., we can enhance ROPFuscator by integrating them as a component in our obfuscation. For this reason, we considered using opaque constants to compute gadget addresses [25].

## VII. Conclusion

We present ROPFuscator, a compiler-driven obfuscation pass based on ROP for any programming language supported by LLVM. Although previous work already explores the effectiveness of ROP as an obfuscation technique, our approach deals with evolving reverse engineering attacks by introducing a unified threat model for ROP-based obfuscation techniques. We introduce a novel instruction hiding technique, later integrated with opaque predicates and constants, to provide a configurable yet robust framework.

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

Finally, we release the source code of ROPFuscator[7] to encourage future research in the systematic hardening of obfuscation schemes.

## VIII. Acknowledgements

The authors want to express their sincere gratitude to John Ericson, Ilya Grishchenko, Francesco Mecca, and Fabio Pagani for their invaluable contributions, guidance, and support throughout the development of this study.

## References

[1] D. Aucsmith, "Tamper resistant software: an implementation," in *Proc. IH '96*, 1996, pp. 317–333.

[2] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *Proc. ISC '01*, 2001, pp. 144–155.

[3] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. CCS '03*, 2003, pp. 290–299.

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Comput. Surv.*, vol. 49, no. 1, 2016.

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

[6] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. POPL '98*, 1998, pp. 184–196.

[7] B. Anckaert, M. Jakubowski, and R. Venkatesan, "Proteus: virtualization for diversified tamper-resistance," in *Proc. DRM '06*, 2006, pp. 47–58.

[8] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Secur. Priv.*, vol. 5, no. 3, pp. 82–84, 2007.

[9] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening code obfuscation against automated attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3055–3073.

[10] N. Eyrolles, "Obfuscation with mixed boolean-arithmetic expressions: reconstruction, analysis and simplification tools," Ph.D. dissertation, Université Paris-Saclay, 2017.

[11] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*. Springer, 2007, pp. 61–75.

[12] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proc. CCS '07*, 2007, pp. 552–561.

[13] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[14] V. Mohan and K. W. Hamlen, "Frankenstein: Stitching malware from benign binaries," in *Proc. WOOT '12*, 2012.

[15] P. Borrello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "The ROP needle: hiding trigger-based injection vectors via code reuse," in *Proc. SAC '19*, 2019, pp. 1962–1970.

[7] https://github.com/ropfuscator/legacy

[16] P. Teuwen, P. Rombouts, J. R. Brands, and J. Hoogerbrugge, "Return-oriented programming as an obfuscation technique," U.S. Patent US9411597B2, May 2014.

[17] D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, and L. Shi, "ROPOB: Obfuscating binary code via return oriented programming," in *Proc. SecureComm '17*, vol. 238, 2018, pp. 721–737.

[18] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. Khurram Khan, "Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions," *J. Netw. Comput. Appl.*, vol. 48, pp. 44–57, 2015.

[19] K. Lu, D. Zou, W. Wen, and D. Gao, "deRop: removing return-oriented programming from malware," in *Proc. ACSAC '11*, 2011, pp. 363–372.

[20] M. Graziano, D. Balzarotti, and A. Zidouemba, "ROPMEMU: A framework for the analysis of complex code-reuse attacks," in *Proc. ASIACCS '16*, 2016, pp. 47–58.

[21] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. S&P '15*, 2015, pp. 674–691.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. S&P '16*, 2016, pp. 138–157.

[23] P. Borrello, E. Coppa, and D. D'Elia, "Hiding in the particles: When return-oriented programming meets program obfuscation," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2021, pp. 555–568.

[24] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in turing-complete return-oriented programming," in *Proc. WOOT '12*, 2012.

[25] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. ACSAC '07*, 2007, pp. 421–430.

[26] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections)," in *Proc. ACSAC '19*, 2019, pp. 177–189.

[27] D. C. D'Elia, E. Coppa, A. Salvati, and C. Demetrescu, "Static analysis of ROP code," in *Proc. EuroSec '19*, 2019, pp. 1–6.

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. USENIX Security '11*, 2011.

[29] A. Tamches and B. P. Miller, "Fine-grained dynamic instrumentation of commodity operating system kernels," in *Proc. OSDI '99*, 1999.

[30] L. Zobernig, S. D. Galbraith, and G. Russello, "When are opaque predicates useful?" in *Proc. TrustCom/BigDataSE '19*, 2019, pp. 168–175.

[31] B. Sheridan and M. Sherr, "On manufacturing resilient opaque constructs against static analysis," in *Proc. ESORICS '16*, vol. 9879, 2016, pp. 39–58.

[32] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[33] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proc. ACSAC '16*, 2016, pp. 189–200.

[34] Q. Liu, R. Safavi-Naini, and N. P. Sheppard, "Digital rights management for content distribution," in *Proc. ACSW Frontiers '03*, vol. 21, 2003, pp. 49–58.

[35] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proc. PLDI '07*, 2007, pp. 89–100.

[36] Y. X. Gu, H. Johnson, C. Liem, A. Wajs, and M. J. Wiener, "White-box cryptography: practical protection on hostile hosts," in *Proc. SSPREW '16*, 2016, pp. 1–8.

[37] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. CCS '15*, 2015, pp. 757–768.

[38] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, and T.-H. Le, "Defeating opaque predicates statically through machine learning and binary analysis," in *Proc. SPRO'19*, 2019, pp. 3–14.

[39] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A verifiable approach to partially-virtualized binary code simplification," in *Proc. CCS '18*, 2018.

[40] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," *Conditionally Accepted to the 42nd IEEE Symposium on Security and Privacy (SP)*, 2021.

[41] D. Andriesse, H. Bos, and A. Slowinska, "Parallax: Implicit code integrity verification using return-oriented programming," in *Proc. DSN '15*, 2015, pp. 125–135.

[42] K. Lu, S. Xiong, and D. Gao, "RopSteg: program steganography with return oriented programming," in *Proc. CODASPY '14*, 2014, pp. 265–272.

[43] S. Drape, "Intellectual property protection using obfuscation," University of Oxford, Tech. Rep. CS-RR-10-02, 2010.

[44] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Proc. ACSAC'00*, 2000, pp. 308–316.