Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation

Pietro Borrello
Sapienza University of Rome
borrello@diag.uniroma1.it

Emilio Coppa
Sapienza University of Rome
coppa@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

***Abstract*—Largely known for attack scenarios, code reuse techniques at a closer look reveal properties that are appealing also for program obfuscation. We explore the popular return-oriented programming paradigm under this light, transforming program functions into ROP chains that coexist seamlessly with the surrounding software stack. We show how to build chains that can withstand popular static and dynamic deobfuscation approaches, evaluating the robustness and overheads of the design over common programs. The results suggest a significant amount of computational resources would be required to carry a deobfuscation attack for secret finding and code coverage goals.**

***Index Terms*—Code obfuscation, program protection, ROP**

I. Introduction

Memory errors are historically among the most abused software vulnerabilities for arbitrary code execution exploits [1]. Since the introduction of system defenses against code injection attempts, code reuse techniques earned the spotlight for their ability in reassembling existing code fragments of a program to build the execution sequence an attacker desires.

Return-oriented programming (ROP) [2] is the most eminent code reuse technique. Thanks to its rich expressivity, ROP has also seen several uses besides exploitation. Researchers have used it constructively, for instance, in code integrity verification [3], or maliciously to embed hidden functionality in code that undergoes auditing [4], [5]. Security firms have reported cases of malware in the wild written in ROP [6].

Some literature considers ROP code bothersome to analyze: humans may struggle with the exoticism of the representation, and the vast majority of tools used for code understanding and reverse engineering have no provisions for code reuse payloads [7]–[10]. Automatic proposals for analyzing complex ROP code started to emerge only recently [7], [9], [10].

We believe that the quirks of the ROP paradigm offer promising opportunities to realize effective code obfuscation schemes. In this paper we present a protection mechanism that builds on ROP to hide implementation details of a program from motivated attackers that can resort to a plethora of automated code analyses. We analyze what qualities make ROP appealing for obfuscation, and address its weak links to make it robust in the face of an adversary that can symbiotically combine general and ROP-aware code analysis methods.

Motivation: From a code analysis perspective, we observe that the control flow of a ROP sequence is naturally destructured. Each ROP gadget ends with a `ret` instruction that operates like a dispatcher in a language interpreter: `ret` reads from the top of the stack the address of the next gadget and transfers control to it. The stack pointer RSP becomes a *virtual program counter* for the execution, sidelining the standard instruction pointer RIP, while gadget addresses become the instructions supported by this custom language.

This level of *indirection* makes the identification of basic blocks and of control transfers between them not immediate. This challenges humans and classic disassembly and decompilation approaches, but may not be an issue for dynamic deobfuscation approaches that explore the program state space systematically (e.g., symbolic execution [11]) or try to extricate the original control flow from the dispatching logic (e.g., [7]), nor for ROP-aware analyses that dissect RSP and RIP changes. Protecting transfers is critical for program obfuscations to withstand advanced deobfuscation methods, and we introduce three ROP transformations that address this weak link.

Another benefit from using ROP for obfuscation is the *code diversity* [12] it can bring. Obfuscations may randomize the instructions emitted at specific points, but can incur a limited transformation space [13]. We can use multiple equivalent gadgets in the encoding to serve one same purpose in different program points. But one gadget can also serve different purposes in different points: the instructions in it that concur to the program semantics will depend on the surrounding chain portion, while the others are dynamically dead. This not only complicates manual analysis, but helps also against pattern attacks that may try to recognize specific gadget sequences to deem the location of ROP branches and blocks in the chain.

Such attacks often complement an attacker's toolbox [14]: for instance, an adversary may heuristically look for distinctive instructions in memory and try to patch away parts that hinder semantic attacks. We identify a distinctive benefit of ROP: the adversary only sees bytes that form gadget addresses or data operands, and because of indirection needs to dereference addresses to retrieve the actual instructions. With a careful encoding we can induce *gadget confusion* that makes it harder also to locate the position of gadget addresses in the chain.

Contributions: In this work we bring novel ideas to the software protection realm, presenting a protection mechanism that significantly slows down or deters current automated deobfuscation attacks. We show how to transform entire program functions into ROP chains that interact seamlessly with standard code components, introducing novel natural encoding transformations that raise the bar for general classes of attacks.

We evaluate our techniques over synthetic functions for two common deobfuscation tasks, putting the computational effort for succeeding into perspective with different configurations of the prominent virtualization obfuscation [13]. We also analyze their slowdowns on performance-sensitive code, and their coverage on a heterogeneous real-world code base. In summary, over the next sections we present:

- a rewriter that turns compiled functions into ROP chains;
- an analysis of ROP in the face of three attack surfaces for general deobufscation, and three encoding predicates that increase the resistance against such attacks;
- a resistance study for secret finding and code coverage goals with symbolic, taint-driven, and ROP-aware tools;
- a coverage study where we transform 95.1% of the unique functions composing the `coreutils` Linux suite.

In the hope of fostering further work in program protection, we make our system available to researchers. Details for access can be found at https://github.com/pietroborrello/raindrop/.

II. Preliminaries

This section details key concepts from code obfuscation and ROP research that are relevant to the ideas behind this paper.

A. Code Obfuscation

Software obfuscation protects digital assets [15] from malicious entities that some literature identifies as MATE (man-at-the-end) attackers [13]. Before a research community was even born, in the '80s these entities challenged and subverted anti-piracy schemes from vendors, and shielded their own malware.

Today it represents an active research area, with heterogeneous protection mechanisms challenged by increasingly powerful program analyses [16]. Data transformations alter the position and representation of values and variables, while code transformations affect the selection, orchestration, and arrangement of instructions. Our focus are code transformations that prevent an adversary from understanding the program logic.

The interpretation capabilities of an attacker can be syntactic, semantic, or both. This distinction makes a great impact: for instance, instruction substitution or the insertion of spurious computations get in the way of syntax-driven attacks, but may hardly affect a semantic interpretation as we discuss in §III. When facing mixed capabilities, the most resilient protection schemes are often heavy-duty transformations that deeply affect the control flow and instructions of a program.

Such transformations commonly operate at the granularity of individual functions [13]. *Control-flow flattening* [17], [18] collapses all the basic blocks of the control-flow graph (CFG) into a single layer, introducing a dispatcher block that picks the next block to execute based on an augmented program state. After the successful deobfuscation attack of [19], present variants try to complicate the analysis of the dispatcher [20].

Virtualization obfuscation [13] completely removes the original layout and instructions: it transforms code into instructions for a randomly generated architecture and synthesizes an interpreter for it [21]. The instructions form a bytecode representation in memory, and the interpreter maintains a

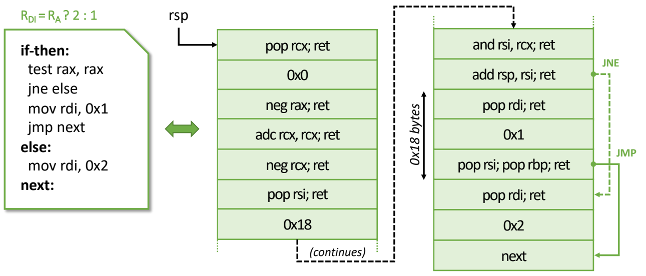
```
if-then:
  test rax, rax
  jne else
  mov rdi, 0x1
  jmp next
else:
  mov rdi, 0x2
next:
```

Fig. 1: ROP chain with non-linear control flow. For readability pointed-to instructions appear in place of gadget addresses.

virtual program counter over it: it reads each instruction and dispatches an *opcode handler* function that achieves the desired semantics for it. As its working resembles a virtual machine, the transformation is also known as VM obfuscation. This technique has lately monopolized the agenda of much deobfuscation research in security conferences (e.g., [8], [22]–[25]). VM obfuscation tools have three main strengths: complex code used in opcode handlers to conceal their semantics, obfuscated virtual program counter updates, and scarce reuse of deobfuscation knowledge as the instruction set and the code for opcode handlers are generated randomly for each program.

Best practices often use data transformations at strategic points (e.g., VM dispatcher) in the implementation of a code transformation. The most common instance are *opaque predicates* [26]: expressions whose outcome is independent of their constituents, but hard to determine statically for an attacker. Opaque predicates can build around mathematical formulas and conjectures, mixed boolean-arithmetic (MBA) expressions, and instances of other hard problems like *aliasing* [27].

B. Return-Oriented Programming

ROP is a technique to encode arbitrary behavior in a program by borrowing and rearranging code fragments, also called gadgets, that are already in the program [2]. Each gadget delivers a piece of the desired computation and terminates with a `ret` instruction, which gives the name to the technique. A ROP payload comprises a sequence of gadget addresses interleaved with immediate data operands. The key to ignition is a pivoting sequence that hijacks the stack pointer, so that on a function return event the CPU fetches the instructions from the first gadget. Each gadget eventually transfers execution to the next using its own `ret` instruction, realizing a ROP chain.

Figure 1 features a chain that assigns register RDI with 1 when register RAX==0, and with 2 otherwise. The example showcases exoticisms of the representation with branch encoding and path-dependent semantics of chain items. The first gadget writes the immediate value `0x0` to RCX, and RSP advances by `0x10` bytes for its `pop` and `ret` instructions. The next two gadgets check if RAX is zero with `neg rax`: the carry flag becomes 0 when RAX==0 and 1 otherwise, then an addition with carry writes this quantity into RCX.

ROP control-flow branches are variable RSP addends computed over a leaked CPU condition flag. The chain determines whether to skip over the `0x18` byte-long portion that sets RDI to 1: it computes in RSI an addend that is equal to 0 when RAX==0, and to `0x18` otherwise, using a two's complement and a bitwise AND on RCX. If the branch is taken, RSP reaches a `pop rdi` gadget that reads and assigns 2 to RDI as desired. When execution falls through, a similar sequence sets RDI to 1, then unconditionally jumps over the alternative assignment sequence: this time we find no RSP addition, but a gadget disposes of the alternative $0x10$ byte-long segment by popping two junk immediates to RSI and RBP.

Attackers can find Turing-complete sets of gadgets in mainstream software [2], [28]. While a few works address automatic generation of ROP payloads [28], publicly available tools often produce incomplete chains in real-world scenarios [29] or do not support branches. Reasons for this failure are side effects from undesired code in found gadgets, register conflicts during chaining [29], and unavailability of "straightforward" gadgets for some tasks [30]. Albeit improved tools continue to appear (e.g., [31]), no general solution for automatic ROP code generation seems available to date.

ROP is the most popular but not the sole realization of code reuse: `jmp`-ended gadgets (JOP) [32], counterfeit C++ objects (COOP) [33], and other elements can be abused as well. But most importantly, ROP today is no longer only a popular mean to get around and disable code injection defenses.

Researchers and threat actors used its expressivity to create userland [6], kernel [34], [35], and enclave [36] malware, and to fool antivirus engines [5], [37] and application review [4]. The sophistication of these payloads went in some cases beyond what a human analyst can manually investigate [9], and researchers in the meantime explored automated approaches to untangle ROP chains: we discuss these works in detail in §III.

III. Adversarial Model

This paper considers a motivated and experienced attacker that can examine a program both statically and dynamically. The attacker is aware of the design of the used obfuscation, but not of the obfuscation-time choices made when instantiating the approach over a specific program to be protected (e.g. at which program locations we applied some transformation).

While the ultimate end goal of a reverse engineering attempt can be disparate, we follow prior deobfuscation literature (e.g., [14], [38], [39]) in considering two deobfuscation goals that are sufficiently generic and analytically measurable:

- G_1 **Secret finding.** The program performs a complex computation on the input, such as a license key validation, and the attacker wishes to guess the correct value;
- G_2 **Code coverage.** The attacker exercises enough (obfuscated) paths to cover all reachable (original) program code, e.g. to later analyze execution traces.

The attacker has access to state-of-the-art systems suitable for automated deobfuscation and can attempt to *symbiotically* combine them, using one to ease another. In the following we describe the most powerful and promising approaches available to attackers, and enucleate three attack surfaces for general deobfuscation. Those will drive our ROP encoding techniques to build chains that may withstand such attacks.

A. Principles behind Automated Deobfuscation

Banescu et al. [38] identify a common pre-requisite in automated attacks perpetrated by reverse engineers: the need for building a suite of inputs that exercise the different paths a protected program can actually take. Achieving a coverage as high as 100% represents G_2 for our attacker, while for G_1 depending on the specific function fewer paths may suffice but also data dependencies should be solved. Slowing down the generation of a "test suite" for the attacker is a first cut of the effectiveness of an obfuscation [38], as it is a key step in most deobfuscation pipelines for utterly disparate end goals.

By analyzing deobfuscation research, we abstract three general attack surfaces that a "good" obfuscation shall consider:

- A_1 **Disassembly.** It should not be immediate for an attacker to discover code portions using static analysis techniques;
- A_2 **Brute-force search.** Syntactic code manipulations such as tracking and "inverting" the direction of control transfers should not reveal new code, but further dependencies must be solved in order to take valid alternate paths;
- A_3 **State space.** When an obfuscation makes provisions to artificially extend the program state space to be explored, analyses based on forward and backward dependencies of program variables should fail to simplify them away.

In the next section we present eminent approaches for such automated attacks, which we then consider in §VII to evaluate our ROP obfuscation. How to transform an existing program function into a ROP chain and make it robust against these three attack types are the subject of §IV and §V, respectively.

B. State-of-the-art Deobfuscation Solutions

1) General Techniques: Deobfuscation attacks can draw from static and dynamic program analyses. Several static techniques are capable of reasoning about run-time properties of the program, and may be the only avenue when the attacker cannot readily bring execution to a protected program portion or control its inputs. In this context, *symbolic execution* (SE) reveals the multiple paths a piece of code may take by making it read symbolic instead of concrete input values, and by collecting and reasoning on path constraints over the symbols at every encountered branch. Upon termination of each path, an SMT solver generates a concrete input to exercise it [11].

Scalability issues often cripple static approaches, and dynamic solutions may try to get around them by leveraging facts observed in a concrete execution. *Dynamic symbolic execution* (DSE) interleaves concrete and symbolic execution, collecting constraints at branch decisions that are now determined by the concrete input values, and generates new inputs by negating the constraints collected for branching decisions.

Obfuscators can however induce constraints that are hard for a solver, or expand the program state space artificially. Building on the intuition that these transformations are not part of the original program semantics, *taint-driven simplification* (TDS) tracks explicit and implicit flows of values from inputs to program outputs, untangling the control flow of an obfuscation method apart from that of the original program [7].

TDS is a general, dynamic, and semantics-based technique: it applies a selection of semantics-preserving simplifications to a recorded trace and produces a simplified CFG. TDS can operate symbiotically with DSE to uncover new code by feeding DSE with the simplified trace: in [7] this symbiosis turned out effective in cases that DSE alone could not handle. TDS succeeded on code protected by state-of-the-art VM obfuscators, as well as on four hand-written ROP programs.

We consider SE, DSE, and TDS as they represent powerful tools available to adversaries, and embody concepts seen also in attacks against specific obfuscations (e.g., [40]). Prior literature [14], [38], [39] uses SE and DSE to evaluate and compare obfuscation techniques on goals akin to G_1 and G_2, as both approaches are powerful and driven only by the semantics of the code (i.e., syntactic changes have little effect).

2) ROP-Aware Techniques: Expressing programs as ROP payloads affects analysis techniques that account for the syntactic representation of code. For instance, even commercial disassemblers and decompilers are not equipped to deal with this exotic representation and would fail to produce meaningful outputs for ROP chains [41]. Currently researchers have come up with two solutions to handle complex ROP code.

ROPMEMU [9] attempts dynamic multi-path exploration by looking for sequences that leak condition flags from the CPU status register, as they may take part in branching sequences (§II-B): it flips their value and tries to generate alternate execution traces that explore new code. ROPMEMU is not the sole embodiment of this technique, seen in, e.g., crash-free binary exploration [42] and malware unpacking [43] research for RIP-driven code. ROPMEMU eventually removes the ROP dispatching logic (i.e., the `ret` sequences) and performs further simplifications, reconstructing a CFG representation.

ROPDissector [10] addresses shortcoming of ROPMEMU in branch identification, with a data-flow analysis for identifying sequences that build variable RSP offsets, so to flip all and only the operations taking part in the process. ROPDissector builds a ROP CFG highlighting branching points and basic blocks in a chain, and operates as a static technique as it does not require a valid execution context as starting point.

In our evaluation we will consider a combination of the two approaches, speculating on extensions tailored to our design.

IV. Program Encoding with ROP

We design a binary rewriter for protecting compiled programs: the user specifies one or more functions of interest that the rewriter encodes as self-contained ROP chains stored in a data section of the binary. Our implementation supports compiler-generated, possibly stripped x64 Linux binaries. To ensure compatibility between ROP chains and non-ROP code modules, we intercept and preserve stack manipulations and use a separate stack for the chain. This section details the design of the rewriter (Figure 2), how it encodes generic functions as self-contained chains, and its present limitations.

A. Geometry of a ROP Encoder

1) Gadget Sources: The first decision to face in the design of a ROP encoder is where to find gadgets. These may be found in statically and dynamically linked libraries, in program parts left unobfuscated, or in custom code added to the program. We ruled out static libraries as a binary might not have any, and dynamic ones to avoid dependencies on specific library versions that must be present in any target system.

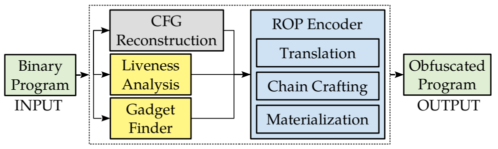

Fig. 2: Architecture of the ROP rewriter.

Exploitation research suggests that program code as small as 20-100KB may already contain minimal gadget sets for attacks [28]. Our scenario however is ideal: the possibility of controlling and altering the binary grants us more wiggle room compared to attack scenarios, as we can add missing gadgets—and most importantly create diversified alternatives—as dead code in the `.text` section of the program. We thus pick gadgets from a pool of artificial gadgets combined with gadgets already available in program parts left unobfuscated.

2) Rewriting: The second decision concerns deploying the encoder as a binary rewriter (as we do) or a compiler pass. Binary rewriting can handle a larger pool of programs, including proprietary software and programs with a custom compilation toolchain, and builds on analyses that extract facts necessary to assist the rewriting. A compiler pass has some such analyses (e.g., liveness) already available during compilation, and possibly more control over code shape. However, in order to be able to rewrite an *entire* function, we believe a pass may have to operate as last step (modifying or directly emitting machine instructions) and/or constrain or rewrite several pieces of upper passes (e.g., instruction selection, register allocation). This would lead to a pass that is platform-dependent and that faces similar challenges to a rewriter while being less general.

3) Control Transfers and Stack Layout: Obfuscated functions get expressed in ROP, but may need to interact with surrounding components, calling (or being called by) non-ROP program/library functions or other ROP functions. In this respect native code makes assumptions on the stack layout of the functions, e.g., when writing return addresses or referencing stack objects in the scope of a function and its callees.

Reassembleable disassembling literature [44]–[47] describes known hurdles when trying to turn hard-coded stack references into symbols that can be moved around. In our design we instead preserve the original stack behavior of the program: we place the chain in a separate region, and rewrite RSP dereferences and value updates to use a `other_rsp` value that mimics how the original code would see RSP (Figure 3).

This choice ensures a great deal of compatibility, and avoids that calls to native functions may overwrite parts of the ROP chain when executing. We keep `other_rsp` in a *stack-switching array* `ss` that ensures smooth transitions between the ROP and native domains and supports multiple concurrently active calls to ROP functions, including (mutual) recursion and interleavings with native calls.

We store the number of active ROP function instances in the first cell of the array, making the last one accessible as `*(ss+*ss)`. When upon a call we need to switch to the native domain, we use `other_rsp` to store the resumption point for the ROP call site, and move its old value in RSP so to switch stacks. Upon function return, a special gadget switches RSP and `other_rsp` again (Figure 4).

4) Chain Embedding: Upon generation of a ROP chain, we replace the original function body in the program with a stub that switches the stack and activates the chain. We opt for chains without destructive side effects, avoiding to have to restore fresh copies across subsequent invocations. We place the generated chains at the end of the executable's `.data` section or in a dedicated one.

B. Translation, Chain Crafting, and Materialization

This section describes the rewriting pipeline we use in the ROP encoder of Figure 2. Although we operate on compiled code, the pipeline mirrors typical steps of compiler architectures [48]: we use a number of support analyses (yellow and grey boxes) and *translate* the original instructions to a simple custom representation made of *roplets*, which we process in the *chain crafting* stage by selecting suitable gadgets for their lowering and then allocating registers and other operands. A final *materialization* step instantiates symbolic offsets in the chain and embeds the output raw bytes in the binary.

1) Translation: The unit of transformation is the function. We identify code blocks and branches in it using off-the-shelf disassemblers (*CFG reconstruction* element of Figure 2): Ghidra [49] worked flawlessly in our tests when analyzing indirect branches, and we support angr [50] and radare2 [51] as alternatives. We then translate one basic block at a time, turning its instructions into a sequence of roplets.

A *roplet* is a basic operation of one of the following kinds:

- *intra-procedural transfer*, for direct branches and for indirect branches coming from switch tables (see [52]);
- *inter-procedural transfer*, for calls to non-ROP and ROP functions (including `jmp`-optimized tail recursion cases);
- *epilogue*, for handling instructions like `ret` and `leave`;
- *direct stack access*, when dereferencing and updating RSP with dedicated read or write primitives (e.g., `push`, `pop`);
- *stack pointer reference*, when the original program reads the RSP value as source or destination operand in an instruction, or alters it by, e.g., adding a quantity to it;
- *instruction pointer reference*, to handle RIP-relative addressing typical of accesses to global storage in `.data`;
- *data movement*, for `mov`-like data transfers that do not fall in any of the three cases above;
- *ALU*, for arithmetic and logic operations.

One roplet is usually sufficient to describe the majority of program instructions. In some cases we break them down in multiple operations: for instance, for a `mov qword [rsp+8], rax` we generate a stack pointer reference and a data movement roplet. To ease the later register allocation step, we annotate each roplet with the list of live registers[1] found for the original instruction via *liveness analysis*.

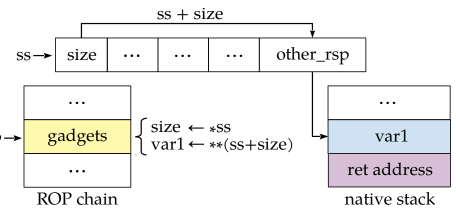

Fig. 3: Reading a stack variable from top of native stack.

At this stage we parametrically rewire every stack-related operation to use `other_rsp`, and transform RIP-relative addressing instances in absolute references to global storage.

2) Chain Crafting: When the representation enters the chain crafting stage, we lower the roplets in each basic block by drawing from suitable gadgets for each roplet type (using the *gadget finder* element of Figure 2). For instance, to translate a conditional (left) or unconditional (right) intra-procedural transfer we combine gadgets to achieve:

```
pop {reg1}   ## L                 ||
mov {reg2}, 0x0                   ||
cmov{ncc} {reg1}, {reg2}          ||   pop {reg1}   ## L
add rsp, {reg1}                   ||   add rsp, {reg1}
```

where a gadget may cover one or more consecutive lines (so we omit `ret` above). In both codes the `pop` gadget will read from the stack an operand `L` (placed as an immediate between the addresses of the first and second gadget) that represents the offset of the destination block. `L` is a symbol that we materialize once the layout of the chain is finalized, similarly to what a compiler assembler does with labels.

Following the analogy, when choosing gadgets for roplets we operate as when in the *instruction selection* stage of a compiler [48], with `{regX}` representing a virtual register, roplets the middle-level representation, and gadgets the low-level one. When it comes to *instruction scheduling*, we follow the order of the original instructions in the block.

Native function calls see a special treatment, as we have to switch stacks and set up the return address in a way to make another switch and resume the chain (§IV-A). For the call we combine gadgets as in the following:

```
pop {reg1}   ## ss
add {reg1}, qword ptr [{reg1}]   ## step A ends
sub qword ptr [{reg1}], 0x8
mov {reg2}, qword ptr [{reg1}]
pop {reg3}   ## addr. of return gadget
mov qword ptr [{reg2}], {reg3}   ## step B ends
pop {reg2}   ## function address
xchg rsp, qword ptr [{reg1}]; jmp {reg2} ## step C
```

where we pop from the stack the addresses of: the stack-switching array, a *function-return gadget*, and the function to call. Gadgets may cover one or more consecutive lines, except for the last one which describes an independent single JOP gadget (§II-B): `xchg` and `jmp` switch stacks and jump into the native function at once. Figure 4 shows the effects of the three main steps carried by the sequence.

[1]A backward analysis deems a register *live* if the function may later read it before writing to it, ending, or making a call that may clobber it [53], [54].

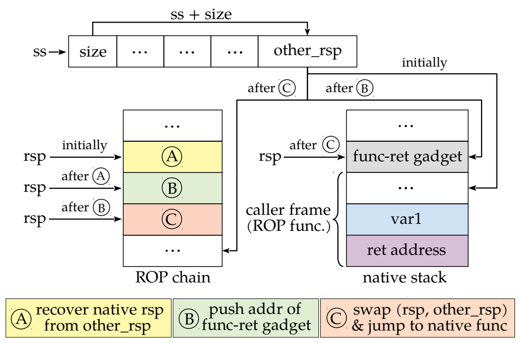

Fig. 4: Call to a native function from ROP code.

The called native function sees as return address (top entry of its stack frame) the address of the function-return gadget. This is a synthetic gadget with a statically hard-wired `ss` address that reads the RSP value saved by the `xchg` at call time and swaps stacks again:

```
mov {reg1}, ss; add {reg1}, qword ptr [{reg1}];
xchg rsp, qword ptr [{reg1}]; ret
```

For space limitations we omit details on the lowering of other roplet types: their handling becomes ordinary once we translated RSP and RIP-related manipulations (§IV-B1).

Register allocation is the next main step: we choose among candidates available for a desired gadget operation by taking into account the registers they operate on and those originally used in the program, trying to preserve the original choices whenever possible. When we find conflicts that may clobber a register, we use scratch registers when available (i.e., non-live ones) or spill it to an inlined 8-byte chain slot as a fallback. We then ensure a *reconciliation of register bindings* [55] at the granularity of basic blocks: when execution leaves a block, the CPU register contents reflect the expected locations for program values that are live in the remainder of the function.

Another relevant detail is to preserve the status register if the program may read it later. While most instructions alter CPU flags, our liveness analysis points out the sole statements that may concur to a later read: whenever in between we introduce gadgets that pollute the flags, we spill and later restore them.

3) Materialization: At the end of the crafting stage a chain is almost readily executable. As its branching labels are still symbolic, we may optionally rearrange basic blocks: then once we fix the layout the labels become concrete RSP-relative displacements. We then embed the chain in the binary, allocating space for it in a data section and replacing the original function code with a pivoting sequence to the ROP chain. The sequence extends the stack-switching array and saves the native RSP value, then the chain upon termination executes a symmetric unpivoting scheme (details in [52]).

C. Discussion

Our design makes limited assumptions on the input code: it hinges on off-the-shelf binary analyses to identify intra-procedural branch targets, and obliviously translates stack accesses and dereferences to preserve execution correctness when interacting with the surrounding software stack.

Our implementation could rewrite a large deal of real-world programs (§VII-C1), even when we supplied it code already protected by the control-flow flattening and/or (nested) VM obfuscations of the Tigress framework [56]. We experimentally observed (§VII-C1) that the analyses of Ghidra are remarkably effective in recovering intra-procedural indirect branch targets, which in several high-level languages derive from optimized switch constructs. Whenever those may fail, one could couple the rewriter with a dynamic tracer for recovering the intended targets by running the original program using expected inputs. Transfers to other functions via indirect calls or tail jumps are instead straightforward, as the chain transfers control to the prologue of the callee as it happens with direct calls.

A limitation of the design, shared with static rewriting and instrumentation schemes [45], [57], is lastly the inability to handle self-modifying and dynamically generated code.

As for register conflicts, the high number of x64 registers give us wiggle room to perform register renaming within blocks with modest spilling. An area larger than the 1-word one we use may help with code with very high register pressure cases (§VII-C) or 32-bit implementations; instruction reordering and function-wide register renaming may also help.

The implementation incurs two main limitations that one can address with moderate effort. The spilling slots and the `ss` array area are not thread-private, but we may recur to thread-local storage primitives. Rewritten binaries are compatible with address space layout randomization for libraries, while the body of the program is currently loaded at fixed addresses. To ship position-independent executables we may add relocation information to headers so to have the loader patch gadget addresses in the chains, or use the online patching for chains from [5] to have the program itself do the update.

In terms of compatibility with ROP defenses of modern operating systems, our context is different to an exploitation one where the program stack gets altered and the choice of gadgets is limited. On Windows, for instance, our stack switching upon API calls would already comply with the RSP range checks of *StackPivot* [58]; our liberty to synthesize gadgets would be decisive against *CallerCheck*, which checks if the instruction preceding an API's return address is a call [58]. We refer to prior work [5] for details. A potential issue, which may require the user to whitelist the program, could be instead coarse-grained defenses that monitor branches [59] or micro-architectural effects [60]. However, those are yet to become mainstream as they face robustness and accuracy issues.

Finally, our readers may question if the use of ROP introduces obvious security risks. An attacker needs a write primitive pointing to a chain in order to alter it. In our protected programs, ROP-encoded parts use write operations only for spilling slots, and those cannot go out of bounds. Non-ROP parts never reference chains in write (or read) operations: an attacker would thus have to search for an arbitrary memory write primitive in such parts. Its presence, however, would be an important source of concern even for the original program. Our implementation also supports the generation of read-only chains, which use a slightly longer spilling machinery.

V. Strengthening ROP Programs

In the furrow of prior works (e.g., [7]–[9]) that highlighted the hindrances from the ROP paradigm to reverse engineering attempts, one could anticipate that the design of §IV may challenge manual deobfuscation and code understanding attempts. The common thread of their observations is that the exoticism of the representation—ROP defines a *weird machine* [61]—disturbs humans when compared to native code. The rewriter makes use of all motivating factors for ROP that we outlined in §I, such as destructured control flow and diversity and reuse of gadgets (including gadget confusion that we describe next).

Quantifying the effectiveness of an obfuscation is however a difficult task, as it depends not only on the available tools, but also on the knowledge of the human operating them [38]. A well-established practice in the deobfuscation literature is to measure the resilience to automated deobfuscation techniques, which in most attack scenarios are the fulcrum of reverse engineering attempts and ease subsequent manual inspections [16].

ROP encoding alone is not sufficient for obfuscation. We find control transfers between basic blocks to be its weak link.

Even when diversifying the used gadget instances, an attacker aware of the design may follow the ROPMEMU approach (§III-B) to spot in an execution trace what gadgets add variable quantities to RSP (thus exposing basic blocks), untangle `ret` instructions from the original control flow of the program, and assemble a dynamic CFG from multiple traces.

Protecting control transfers is equally critical in the face of the most effective general-purpose semantics-aware techniques like SE, DSE and TDS, which try to reason on the parts essential for program functionality while sifting out the irrelevant obfuscation constructs and instructions [7], [38], such as side effects and dynamically dead portions from gadgets.

One way to hinder the automated approaches of §III-B would be to target weaknesses of each technique individually. For instance, researchers proposed hard-to-solve predicates for SE (e.g. MBA expressions [62], cryptographic functions [63]), and code transformations that impact concolic variants like DSE too [22]. But an experienced attacker can symbiotically combine methods to defeat this approach, for instance using TDS or similar techniques (e.g., program synthesis for MBA predicates [62]) to feed DSE with tractable traces as in [7].

In this section instead we present three rewrite predicates, naturally meshed with RSP update actions, that bring protection against generic, increasingly powerful automated attacks that cover the principled classes $A_{1\text{-}2\text{-}3}$ from §III-A. We then introduce gadget confusion and share some general reflections.

A. Predicate P_1: Anti-ROP-Disassembly

Our first predicate uses an array of opaque values [64] to hide branch targets (A_1). The array contains seemingly random values generated such that a periodic invariant holds, and backs the extraction of a quantity a that we use to compute the displacement in the chain for one of the n branches in the code. Suppose we need to extract a for branch $b \in \{0..n-1\}$: starting with cell b, in every p-th cell of the array we store a random number q such that $q \equiv a \bmod m$, with $m > n$ and p chosen at obfuscation time.

10	19	34	45	54	62	66	33	6	59	61	20

Above we encoded information for n=3 branches using p=4 repetitions and m=7. For the branch with ordinal 1 we wanted to memorize $a = 5$: every cell colored in dark gray thus contains a value v such that $v \bmod 7$ equals 5.

During obfuscation we use a period of size $s > n$, with a fraction of the cells containing garbage. We also share a valid cell among multiple branches, so to avoid encoding unique offsets that may aid reversing. To this end we divide an RSP branch offset δ in a fixed part a encoded in the array and a branch-specific part $\delta - a$ computed by the chain, then we compose them upon branching.

This implies that for static disassembly an attacker should recover the array representation and mimic the computations made in every chain segment to extract a and compute the branch-specific part. While this is possible for a semantically rich static technique like SE, periodicity comes to the rescue as it brings *aliasing*: every p-th cell is suitable for extracting a. Our array dereferencing scheme takes the form of:

$$a = A[f(x) * s + n] \; mod \, m$$

where $f(x)$ depends on the program state and returns a value between 0 and $p - 1$. Its implementation opaquely combines the contents of up to 4 registers that hold input-derived values. SE will thus explore alternative input configurations that ultimately lead to the same rsp $+= \delta$ update; reducing their number by constraining the input would lead instead to missing later portions of program state.

Whenever an attacker may attempt a *points-to* analysis [65] over rsp $+= \delta$, we believe a different index expression based on user-supplied or statically extracted facts on input value ranges would suffice to complicate such analysis significantly.

B. Predicate P_2: Preventing Brute-Force Search

Our second predicate introduces artificial data dependencies on the control flow, hindering dynamic approaches for brute-force path exploration (A_2) that flip branches from an execution trace. While these techniques do not help in secret finding (G_1) as they neglect data constraints (§III-A), they may be effective when the focus is code coverage (G_2).

P_1 is not sufficient against A_2: an attacker can record a trace that takes a conditional branch shielded by P_1, analyze it to locate the flags set by the instruction that steered the program along the branch, flip them, and reveal the other path [10].

Without loss of generality, let us assume that a *cmp a, b* instruction determines whether the original program should jump to location `L` when $a == b$ and fall through otherwise. We introduce a data dependency that breaks the control flow when brute-force attempts leave its operands untouched. As we translate the branch in ROP, in the block starting at `L` we manipulate RSP with, e.g., *rsp* $+= x * (a - b)$, so that when brute-forcing it without changing the operands, $(a - b) \mathrel{!}= 0$ and RSP flows into unintended code by some offset multiple of x. Similarly, on the fall-through path we manipulate RSP with, e.g., $rsp \mathrel{+}= x * (1 - \mathit{notZero}(a - b))$, where *notZero* is a flag-independent computation[2] so the attacker cannot flip it.

Different formulations of opaque updates are possible. Whenever an attacker may attempt to learn and override updates locally, we figured a future, more covert P_2 variant that encodes offsets for branches using opaque expressions based on value invariants (obtainable via value set analysis [66]) for some variable that is defined in an unrelated CFG block.

C. Predicate P_3: State Space Widening

Our third predicate brings a path-oriented protection that artificially extends the program space to explore and is coupled with data (and optionally control) flows of the program, so that techniques like TDS (A_3) cannot remove it without knowledge of the obfuscation-time choices. P_3 comes in two variants.

The first variant is an adaptation of the FOR predicate from [14]. The idea is to introduce state forking points using loops, indexed by input bytes, that opaquely recompute available values that the program may use later. In its simplest formulation, FOR replaces occurrences of an input value `char c` with uses of a new `char fc` instantiated by `for (i=0; i<c; ++i) fc++`. Such loop introduces 2^8 artificial states to explore due to the uncertainty on the value of `c`.

The work explains that targeting 1-byte input portions brings only a slight performance overhead, and choosing independent variables for multiple FOR instances optimizes composition for state explosion. It also argues how to make FOR sequences resilient to pattern attacks, and presents a theorem for robustness against taint analysis and backward slicing, considered for forward and backward code simplification attacks, respectively (the TDS technique we use has provisions for both [7]).

While we refer to it for the formal analysis, for our goals suffice it to say that when the obfuscated variable is input-dependent (for tainting) and is related to the output (for slicing), such analyses cannot simplify away the transformation.

During the rewriting we use a data-flow analysis to identify which live registers contain input-derived data (*symbolic* registers) and may later concur to program outputs[3]. We then introduce value-preserving opaque computations like in the examples below (the right one is adapted from [14]):

```
// clear last byte
dead_reg &= 0xAB00;
for (i=0; i<(char)sym; ++i)
  dead_reg++;
sym |= (char)dead_reg;
```

```
dead_reg = 0;
for (i=0; i<(char)sym; ++i)
   if (i%2) dead_reg--;
   else dead_reg+=3;
if (i%2) dead_reg-=2;
sym = (sym&0xF..F00)+dead_reg;
```

These patterns significantly slow down SE and DSE engines, but also challenge approaches that feed tractable simplified traces to DSE. While one may think of detecting and propagating constant values in the trace, the TDS paper [7] explains that doing it indiscriminately may oversimplify the program: in our scenario it may remove FOR but also pieces of the logic of the original program elsewhere. To avoid oversimplification the TDS authors restrict constant propagation across input-tainted conditional jumps, which is exactly the case with `dead_reg` and `sym` in the examples above.

The authors suggest, as a general way to hamper semantics-based deobfuscation approaches like TDS, to deeply entwine the obfuscation code with the original input-to-output computations. They also state that at the time obfuscation tools had not explored this avenue, possibly for the difficulties in preserving observable program behavior [7].

Our second P_3 variant is new and moves in this direction. Instead of recomputing input-derived variables, we use them to perform *opaque updates* to the array used by P_1. Updates include adding/subtracting quantities multiple of m, swapping the contents of two related cells from different periods, or combining the contents of two cells i and j where $a \equiv A[i] \bmod m$ and $b \equiv A[j] \bmod m$ to update a cell l where $(a + b) \equiv A[l] \bmod m$. For DSE-alike path exploration approaches the effect is tantamount to the FOR transformation described above. For trace simplification it introduces implicit flows, with *fake control dependencies* between program inputs and branch decisions taken later in the code: TDS cannot simplify them without explicit knowledge of the invariants.

D. Gadget Confusion

ROP encoding brings several advantages when implementing $\mathsf{P}_{1\text{-}2\text{-}3}$. Firstly, it offers significant leeway for diversifying the gadget instances we use to instantiate them. We combine this diversity with *dynamically dead* instructions: we can use gadgets whose each instruction either concurs to implementing a predicate or has no effect depending on the surrounding chain portion. This helps in instantiating many variants of a pattern, challenging syntactic attacks aware of the design.

However, a unique advantage of ROP, as we observed in §I, is the level of indirection that it brings: this complicates pattern attacks that look for specific instruction bytes, since code is not in plain sight, and attackers need to extract the instruction sequences as if executing the program. What they see are bytes belonging to either gadget addresses or data operands. They may, however, attempt analyses that look for byte sequences resembling addresses from code regions (i.e., plausible gadgets) and try to speculatively execute the chain from there [10], [67]. By trying it at every plausible point, this may eventually reveal some chain portions, nonetheless short thanks to $\mathsf{P}_{1\text{-}2}$.

This is when *gadget confusion* enters the picture. Firstly, we can transform data operands in the chain to look like gadget addresses, having then gadgets recover the desired values at run time (e.g., subtracting two addresses to obtain a constant, applying bitmasks, shifting bits, etc.). This is possible as we control both the layout of the binary (for the addresses) and the pool of artificial gadgets (for the manipulations). Now that virtually every 8-byte chain stride looks like a gadget address, we introduce unaligned RSP updates at random program points, adding a quantity η s.t. $\eta \bmod 8 \mathrel{!}= 0$. In the end, the attacker may have to execute speculatively at every possible chain offset, obtaining instructions that may or may not be part of the intended execution sequence. We believe such gadget confusion makes pattern attacks on our chains even harder.

[2]Example: $\mathit{notZero}(n) := \sim(\sim n \,\&\, (n + \sim 0)) \gg 31$ for 32-bit data types.

[3]To this end we use the symbolic execution capabilities of angr [50].

E. Further Remarks

The instantiation of $\mathsf{P}_{1\text{-}2\text{-}3}$ is naturally entwined with RSP dispatching: directly for $\mathsf{P}_{1\text{-}2}$, and indirectly for P_3 through array updates. In the rewriter, P_1 replaces the RSP update sequence we showed in §IV-B2, while P_2 operates on the fall-through and target blocks of a branch. Finally, the rewriter can apply either P_3 variant to a user-defined fraction k of the original program points when lowering the associated roplets.

Each predicate targets a main attack surface, but positive externalities are also present. P_2 can protect against possible linear/recursive disassembly algorithms for ROP (A_1), but will not withstand SE-based disassembly. In §VII-C we discuss how P_1 can slow down state exploration (A_3) by indirectly putting pressure on the memory model of a SE or DSE engine.

Finally, with the second P_3 variant we used ROP control transfer dynamics to introduce also fake control dependencies.

VI. Other Related Works

Prior research explored ROP for software protection goals orthogonal to obfuscation: tamper checking of selected code regions through chains that use gadgets from such regions [3], covert watermark encoding [68], and steganography of short code [69]. Each of them could complement our design, especially [3] for checking code integrity of non-obfuscated parts.

ROPOB [41] is a lightweight obfuscation method to rewrite transfers between CFG basic blocks using ROP gadgets. It considers standard disassembly algorithms as adversary (a "lighter" A_1 case), and does not withstand static attacks like SE (A_1) or ROPDissector (A_2), nor dynamic ones like DSE or TDS (A_3). ROPOB leaves data manipulation instructions in plain sight, whose rewriting poses several challenges (§IV-B).

VM deobfuscation attacks like Syntia [8] and VMHunt [40] intercept and simplify (A_3) dispatching and opcode handling sequences. They do not apply directly to ROP chains, and embody flavors of the agnostic and general approach of TDS.

movfuscator [70] is an extreme instance of the weird machine concept, rewriting programs using only the Turing-complete `mov` instruction. Kirsch et al. present [71] a custom linear-sweep algorithm (A_1) that recovers the CFG by targeting logic dispatching elements used for the very encoding.

VII. Evaluation

We arrange our experimental analysis in three parts. We first study the efficacy of our techniques against prominent solutions for $\mathsf{A}_{1\text{-}2\text{-}3}$ (§VII-A), confirming the theoretical expectations. We then study the resource usage of viable deobfuscation attacks using a methodology adopted in previous works [14], [38], and put such numbers into perspective with VM-obfuscated[4] counterparts (§VII-B). Finally, we analyze the applicability of our method to real-world code (§VII-C).

We ran the tests on a Debian 9.2 server with two Xeon E5-4610v2 and 256 GB of RAM. Our online technical report [52] details the settings we used to generate our 72 test functions and the VM variants with Tigress, and more implementation details. The rewriter currently consists of ~3K Python LOC.

Setting	Description
ROP_k	ROP obfuscation with P_3 inserted at a fraction of program points $k \in \{0, 0.05, 0.25, 0.50, 0.75, 1.00\}$ and with P_1 instantiated with $n{=}4$, $s{=}n$, $p{=}32$
nVM	n layers of VM obfuscation with $n \in \{1, 2, 3\}$
$n\mathrm{VM\text{-}IMP}_x$	n layers of VM obfuscation with implicit flows used for every VPC load at layer(s) $x \in$ {first, last, all}

TABLE I: Terminology for obfuscation configurations.

Table I details configuration naming for the main ROP and VM experiments. For the latter we try multiple layers of nested virtualization as this is known to slow down SE and DSE-based attacks [14], [23], and use a Tigress predicate that adds implicit flows to virtual program counter (VPC) loads: those frustrate taint analysis-based simplifications and also create many redundant states whenever VPC becomes symbolic.

A. Efficacy of ROP Strengthening Transformations

The techniques presented in §V should intuitively raise the bar to existing automated attacks, and hinder symbiotic combinations between them. We now study how each automated approach feels the effects of each technique individually already on small program instances, discussing also design-aware enhancements we tried for ROP tools. In the end, DSE emerges as the one and only viable option for our attacker.

We leverage the Tigress framework [56] to generate functions appropriate as reverse engineering targets with a desired complexity and structure. Tigress will also annotate CFG split and join points with probes to help us measure code coverage.

1) ***General Attacks:*** In the context of general-purpose automated attacks, we consider angr [50] as SE engine, S2E [72] for DSE, and the TDS implementation released by its authors. Let us start with SE. For P_1 we consider a function with control structure [56] `for (if (bb 4) (bb4))` having 4 mathematical computations per block, 15 loop iterations, and a single `int` as input. In a "ROP-P_1" version we encode in the array for P_1 $n{=}4$ δ-offsets, with no garbage entries ($s=n$) and $p=32$ repetitions, for a total of 128 cells populated statically.

To explore enough paths to hit all coverage points (G_2), angr took a time in the order of seconds for the native function, and over 4500 seconds for ROP-P_1. The aliasing P_1 induces on RSP updates for branching slows angr down significantly already for little code, as the SMT solver sees increasingly complex expressions over RSP. Aliasing reverberates on secret finding (G_1) too: with a simpler `for (for (bb 4))` code, angr cracked the secret in the order of seconds for the original code, and over 5 hours for ROP-P_1. Other configurations of variable complexity confirmed these trends. When we tested P_3 shielding a single program point per basic block, 24 hours were not sufficient for angr to crack the secret. These results suggest SE may not be readily suitable against our approach.

As for DSE, in the experiments P_1 impacted it slightly and only for G_2: the reason is that S2E benefits from concrete

[4]We do not consider commercial tools like VMProtect for two reasons: they offer little control over the transformations (but may rather combine many at once), and add tricks and bombs [7] to break deobfuscation solutions by targeting implementation gaps instead of their methodological shortcomings.

input values when picking the next path to execute. For P_3 we obtained two confirmations: its two variants bring similar time increases, and while higher k fractions of shielded program points inflate the state space possibly more, code with small input space may not always offer sufficient independent sources (i.e., symbolic registers) for optimal composition of P_3 instances. We postpone a detailed analysis of the induced overheads to §VII-B as we consider larger code instances.

P_1 and P_3 resist TDS by design. The tested output traces kept non-simplifiable (§V-C) implicit control dependencies from having a tainted input value determine a jump target: as those are pivotal to put pressure on DSE, combining DSE with TDS-simplified input traces [7] would not ease attacks.

Summarizing, P_1 and P_3 effectively raise the bar for A_1 and A_3 attacks, respectively: SE and TDS look no longer useful already for little code. P_2 and gadget confusion target syntactic approaches, unlike the semantics-aware attacks we considered above: we address them next in the ROP-aware domain.

*2) **ROP-Aware Attacks:*** To analyze ROP payloads we use and extend ROPDissector to start from a memory dump of the program taken when entering the chain of interest: in this configuration it operates as a hybrid static-dynamic analysis and surpasses ROPMEMU in branch analysis and flipping capabilities. With ROPDissector now embodying a full-fledged ROP-A_2 approach, we test if it can help with G_2, while G_1 is out of scope as A_2 recovers code but neglects data constraints.

Backing our expectations, shielding branches with P_2 in the rewriting makes ROPDissector fail in revealing any basic block other than those the input used for the test reveals. We tried to further extend ROPDissector by using its gadget guessing technique (a ROP-educated form of pattern matching [10]) to reveal new blocks by executing the chain at different start offsets. Our gadget confusion however makes such analysis explode, with many short and unaligned candidate blocks that are difficult to distinguish from P_2-protected true positives.

We conclude this part by stressing the importance of conceiving all of our protections. P_1 impacts ROPDissector only if no dump is supplied, and P_3 does not affect it directly. Hence, without P_2 an attacker could have used ROPDissector or a similar tool to aid semantic attacks in code coverage scenarios.

B. Measuring Obfuscation Resilience

We now measure the amount of resources required to carry automated attacks for secret finding (G_1) and code coverage (G_2) over synthetic functions from an established methodology. We ask Tigress to generate 72 non-cryptographic hash functions with 6 control structures analogous to the most complex ones from an influential obfuscation work [39], input sizes of $\{1, 2, 4, 8\}$ bytes, and three seeds (details in [52]).

We exclude techniques that were ineffective on smaller inputs like SE, and restrict our focus to DSE (recently [14] makes a similar choice). DSE allows us to set up controlled and accurate experiments for measuring G_1 and G_2, as S2E typically succeeds in either goal in about one minute for each of the 72 functions. This makes measuring obfuscation overheads feasible, with a 1-hour budget per experiment

Obfuscation Configuration	Secret Finding: Found	Secret Finding: Avg Time	Code Coverage: 100% Points
Native	70/72	65.2s	72/72
$ROP_{0.05}$	19/72	907.9s	34/72
$ROP_{0.25}$	10/72	568.4s	11/72
$ROP_{0.50}$	9/72	884.0s	9/72
$ROP_{0.75}$	5/72	775.3s	7/72
$ROP_{1.00}$	1/72	3028.7s	6/72
1VM-IMP$_{all}$	61/72	85.8s	68/72
2VM	62/72	71.6s	67/72
2VM-IMP$_{first}$	62/72	100.4s	66/72
2VM-IMP$_{last}$	61/72	104.1s	65/72
2VM-IMP$_{all}$	62/72	160.6s	64/72
3VM	62/72	119.2s	69/72
3VM-IMP$_{first}$	54/72	899.2s	56/72
3VM-IMP$_{last}$	62/72	240.3s	61/72
3VM-IMP$_{all}$	0/72	-	0/72

TABLE II: Successful attacks in the 1h-budget per program.

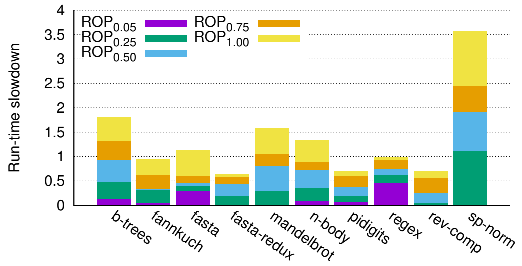

Fig. 5: Run-time overhead for `clbg` benchmarks of different ROP_k settings with 2VM-IMP$_{last}$ used as baseline.

sufficient to capture a slowdown of ~50x or higher. With 15 configurations and 2 goals, the tests took > 2000 CPU hours.

In light of all the considerations made in §VII-A, we use a ROP_k setup with P_1 and P_3 enabled (P_2 and gadget confusion are disabled as they do not affect DSE), with the same $\{s, n, p\}$ settings mentioned there for P_1, and with P_3 instantiated in its first variant and applied at different fractions k of program points (Table I). As state exploration strategy for S2E we use class-uniform path analysis [73] as it consistently yielded the best results across all ROP and VM configurations: its state grouping seems to work effectively for reducing bias towards picking hot spots involved in path explosion, which could be the case with P_3 instances under other strategies.

*1) **Secret Finding:*** Column two and three of Table II summarize the results for the differently obfuscated configurations: for each class we report for how many functions S2E found the secret, and the average time for successful attempts. For 2 of the 72 non-obfuscated functions S2E failed also with a 3-hour budget, likely due to excessively complex path constraints.

Coherently with insights from previous works [14], [23], applying one or two layers of VM obfuscation does not prevent S2E from solving the majority of the secrets (the same sets of 61-62 functions over 72) even when using implicit VPC loads[5], with average overheads as high as 1.6x when applied to either the inner or the outer VPC, and 2.46x when to both. For 3VM implicit VPC loads are significantly more effective in slowing down S2E when applied on the innermost VPC than on the outermost one, while when used at all the three

[5]We do not report data for 1VM and $ROP_{k=0}$ programs since S2E breaks them with no appreciable slowdown w.r.t. their non-obfuscated counterparts.

layers S2E found zero secrets within the 1-hour budget.

The fraction of successful attacks to ROP$_k$ is lower than for VM configurations already for $k = 0.05$, except for 3VM-IMP$_{all}$ that however, as we see in §VII-C2, may bring a destructive impact on program running time. The fraction of ROP$_k$-protected functions that S2E can crack decreases with k: while we cannot compare average times computed for different sets, individual figures reveal that S2E can crack only the simpler functions as k increases, with a higher processing time compared to when they were cracked for a smaller k.

*2) **Code Coverage:*** The last column of Table II lists for how many functions S2E covered all the CFG split and join points annotated by Tigress and reachable in the native counterparts (as the functions are relatively small, we consider coverage an "all or nothing" goal like in [14], [38]). As seen in §III-A, we recall that secret finding may not require full coverage (neither achieving G_2 is sufficient for G_1). For most VM configurations, the functions for which S2E fully explored the original CFG are slightly more than those for which it recovered the secret. ROP$_k$ already for $k = 0.05$ impedes achieving G_2 for nearly half of the functions, and leaves only a handful (6-11) within the reach of S2E for higher k values.

C. Deployability

To conclude our evaluation, we investigate how our methods can cope with real-world code bases in three respects: efficacy of the rewriting, run-time overhead for CPU-intensive code, and an obfuscation case study on a popular encoding function.

*1) **Coverage:*** We start by assessing how our implementation can handle the code base of the `coreutils` (v8.28, compiled with gcc 6.3.0 `-O1`). Popular in software testing, this suite is a suitable benchmark thanks to its heterogeneous code patterns. Using symbol and size information, we identify 1354 unique functions across its corpus of 107 programs. We skip the 119 functions shorter than the 22 bytes the pivoting sequence requires (§IV-B3). Our rewriter could transform 1175 over 1235 remaining functions (95.1%, or a 0.801 fraction if normalized by size). 40 failures happened during register allocation as one spilling slot was not enough to cope with high pressure (§IV-C), 19 for code like `push rsp` and `push qword [rsp + imm]` that the translation step does not handle yet (§IV-B1), and 1 for failed CFG reconstruction.

As informal validation of functional correctness, we run the test suite of the `coreutils` over the obfuscated program instances, obtaining no mismatches in the output they compute.

*2) **Overhead:*** Albeit a common assumption is that heavy-duty obfuscation target one-off or infrequent computations, we also seek to study performance overhead aspects. We consider the `clbg` suite [74] used in compiler research to benchmark the effects of code transformations (e.g., [75], [76]). As a reference we consider 2VM-IMP$_{last}$ as it was the fastest configuration for double virtualization with implicit VPC loads (1VM is too easy to circumvent, and 3VM brings prohibitive overheads, i.e., over 5-6 orders of magnitude in our tests).

Figure 5 uses a stacked barchart layout to present slowdowns for ROP$_k$, as its overhead can only grow with k. With the exception of `sp-norm` that sees repeated pivoting events from a ROP tight loop calling a short-lived ROP subroutine, ROP$_k$ is consistently faster than 2VM-IMP$_{last}$ for $k \leq 0.5$, and no slower than 1.81x (`b-trees` that repeatedly calls `malloc` and `free`) when in the most expensive setting $k = 1.00$.

*3) **Case Study:*** Finally, we study resilience and slowdowns of selected obfuscation configurations on the reference implementation of the popular base64 encoding algorithm [77]. base64 features byte manipulations and table lookups relevant for transformation code of variable complexity that users may wish to obfuscate. An important consideration is that in the presence of table lookups, using concrete values for input-dependent pointers is no longer effective (but even counter-productive) for DSE to explore relevant states. We thus opt for the per-page theory-of-arrays ([11], [78]) memory model of S2E. This choice allows S2E to recover a 6-byte input in about 102 seconds for the original implementation, 180 for 2VM-IMP$_{last}$, 281 for 2VM-IMP$_{all}$, and 1622 for 3VM-IMP$_{last}$.

A budget of 8 hours was not sufficient for 3VM-IMP$_{all}$, as well as for ROP$_k$ already for $k = 0$ (when only P_1 is enabled). As anticipated in §V-E, the aliasing from P_1 on RSP updates can impact the handling of memory in DSE executors in ways that the synthetic functions of §VII-B did not (as they do not use table lookups). As for code slowdowns, ROP$_k$ seems to bring rather tolerable execution times: for a rough comparison, execution takes 0.299ms for ROP$_{0.25}$ and 1.791ms for ROP$_{1.00}$, while for VM settings we measured 1.63ms for 2VM-IMP$_{last}$, 347ms for 2VM-IMP$_{all}$, 668ms for 3VM-IMP$_{last}$ and 2211s for the unpractical 3VM-IMP$_{all}$.

VIII. Concluding Remarks

Adding to the appealing properties of ROP against reverse engineering that we discussed throughout the paper, the experimental results lead us to believe that our approach can:

1) hinder many popular deobfuscation approaches, as well as symbiotic combinations aimed at ameliorating scalability;
2) significantly increase the resources needed by automated techniques that remain viable, with slowdowns $\geq$ 50x for the vast majority of the 72 targets for both end goals $G_{1\text{-}2}$;
3) bring multiple configuration opportunities for resilience (and overhead) goals to the program protection landscape.

While obfuscation research is yet to declare a clear winner and automated attacks keep evolving, our technique is also orthogonal to most other code obfuscations, meaning it can be applied on top of already obfuscated code (§IV-C). We have followed established practices [13] of analyzing our obfuscation individually and on function units, yet in future work we would like to expand both points: namely, studying mutually reinforcing combinations with other obfuscations, and applying ROP rewriting inter-procedurally, removing the stack-switching step during transfers between ROP functions, since our design allows that. Finally, to optimize composition of symbolic registers when instantiating P_3, we may look at def-use chains as suggested by [14] for FOR cases, exploring analyses like [66] necessary to obtain the required information.

References

[1] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Research in Attacks, Intrusions, and Defenses*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106. [Online]. Available: https://doi.org/10.1007/978-3-642-33338-5_5

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of the 14th ACM Conf. on Comp. and Comm. Sec.*, ser. CCS '07, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[3] D. Andriesse, H. Bos, and A. Slowinska, "Parallax: Implicit code integrity verification using return-oriented programming," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15, 2015, pp. 125–135. [Online]. Available: https://doi.org/10.1109/DSN.2015.12

[4] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When benign apps become evil," in *Proceedings of the 22nd USENIX Security Symposium*, ser. SEC '13, 2013, pp. 559–572.

[5] P. Borrello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "The ROP needle: Hiding trigger-based injection vectors via code reuse," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '19, 2019, pp. 1962–1970. [Online]. Available: https://doi.org/10.1145/3297280.3297472

[6] FireEye, "The Number of the Beast," https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html, 2013, online; accessed 11 June 2020.

[7] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symp. on Sec. and Privacy*, ser. SP '15, 2015, pp. 674–691. [Online]. Available: https://doi.org/10.1109/SP.2015.47

[8] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *Proc. of the 26th USENIX Security Symposium*, ser. USENIX Security 17, 2017, pp. 643–659.

[9] M. Graziano, D. Balzarotti, and A. Zidouemba, "ROPMEMU: A framework for the analysis of complex code-reuse attacks," in *Proceedings of 11th Asia Conference on Computer and Communications Security*, ser. ASIACCS '16, 2016, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897894

[10] D. C. D'Elia, E. Coppa, A. Salvati, and C. Demetrescu, "Static Analysis of ROP Code," in *Proceedings of the 12th European Workshop on Systems Security*, ser. EuroSec '19, 2019. [Online]. Available: http://doi.acm.org/10.1145/3301417.3312494

[11] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computer Surveys*, vol. 51, no. 3, pp. 50:1–50:39, 5 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[12] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014, pp. 276–291. [Online]. Available: https://doi.org/10.1109/SP.2014.25

[13] S. Banescu and A. Pretschner, "Chapter five - A tutorial on software obfuscation," *Advances in Computers*, vol. 108, pp. 283–353, 2018. [Online]. Available: https://doi.org/10.1016/bs.adcom.2017.09.004

[14] 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. of the 35th Annual Comp. Security Applications Conference*, ser. ACSAC '19, 2019, pp. 177–189. [Online]. Available: https://doi.org/10.1145/3359789.3359812

[15] S. R. Subramanya and B. K. Yi, "Digital rights management," *IEEE Potentials*, vol. 25, no. 2, pp. 31–34, 2006.

[16] 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 Computer Surveys*, vol. 49, no. 1, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2886012

[17] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, ser. DSN '01, 2001, pp. 193–202. [Online]. Available: https://doi.org/10.1109/DSN.2001.941405

[18] S. Chow, Y. X. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *Proceedings of the 4th International Conference on Information Security*, ser. ISC '01, vol. 2200, 2001, pp. 144–155. [Online]. Available: https://doi.org/10.1007/3-540-45439-X_10

[19] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Proc. of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05, 2005, pp. 45–54. [Online]. Available: https://doi.org/10.1109/WCRE.2005.13

[20] B. Johansson, P. Lantz, and M. Liljenstam, "Lightweight dispatcher constructions for control flow flattening," in *Proc. of the 7th Software Security, Protection, and Reverse Engineering Workshop*, ser. SSPREW-7, 2017. [Online]. Available: https://doi.org/10.1145/3151137.3151139

[21] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, ser. WCRE '12. USA: IEEE Computer Society, 2012, pp. 61–70. [Online]. Available: https://doi.org/10.1109/WCRE.2012.16

[22] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 732–744. [Online]. Available: https://doi.org/10.1145/2810103.2813663

[23] J. Salwan, S. Bardin, and M. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *Proc. of the 15th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '18, vol. 10885, 2018, pp. 372–392. [Online]. Available: https://doi.org/10.1007/978-3-319-93411-2_17

[24] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proc. of the 30th IEEE Symposium on Security and Privacy*, ser. SP '09, 2009, pp. 94–109. [Online]. Available: https://doi.org/10.1109/SP.2009.27

[25] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011, pp. 275–284. [Online]. Available: https://doi.org/10.1145/2046707.2046739

[26] C. S. Collberg, C. D. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Tech. Rep., 1997.

[27] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. on Prog. Lang. and Sys.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC '11. USENIX Association, 2011.

[29] M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci, "ROPMate: Visually Assisting the Creation of ROP-based Exploits," in *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, ser. VizSec '18, 2018. [Online]. Available: https://doi.org/10.1109/VIZSEC.2018.8709204

[30] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. on Inf. and Sys. Sec.*, vol. 15, no. 1, 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[31] A. Wailly, A. Souchet, J. Salwan, A. Verez, and T. Romand, "Automated Return-Oriented Programming Chaining," https://github.com/awailly/nrop, 2014, online; accessed 11 June 2020.

[32] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, 2011, pp. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[33] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *2015 IEEE Symp. on Sec. and Priv.*, ser. SP '15, 2015, pp. 745–762. [Online]. Available: https://doi.org/10.1109/SP.2015.51

[34] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 2009, pp. 383–398.

[35] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert, "Persistent data-only malware: Function hooks without code," in *21st Annual Network and Distributed System Security Symposium*, ser. NDSS '14, 2014.

[36] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with Intel SGX," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '19, 2019, pp. 177–196. [Online]. Available: https://doi.org/10.1007/978-3-030-22038-9_9

[37] C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis, "Transforming malicious code to rop gadgets for antivirus evasion," *IET Inform. Security*, vol. 13, no. 6, pp. 570–578, 2019. [Online]. Available: https://doi.org/10.1049/iet-ifs.2018.5386

[38] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. ACM, 2016, pp. 189–200. [Online]. Available: https://doi.org/10.1145/2991079.2991114

[39] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning," in *Proc. of the 26th USENIX Security Symposium*, 2017, pp. 661–678.

[40] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A verifiable approach to partially-virtualized binary code simplification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018, pp. 442–458. [Online]. Available: https://doi.org/10.1145/3243734.3243827

[41] D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, and L. Shi, "ROPOB: Obfuscating Binary Code viaReturn Oriented Programming," in *Security and Privacy in Communication Networks*, ser. SecureComm '17. Springer International Publishing, 2018, pp. 721–737.

[42] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-executing binary programs for security applications," in *Proc. of the 23rd USENIX Conf. on Security Symp.*, ser. SEC '14, 2014, pp. 829–844.

[43] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Rambo: Run-time packer analysis with multiple branch observation," in *Proc. of the 13th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '16, 2016, pp. 186–206. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_10

[44] S. Wang, P. Wang, and D. Wu, "UROBOROS: instrumenting stripped binaries with static reassembling," in *IEEE 23rd Int. Conf. on Soft. Analysis, Evol., and Reengineering*, ser. SANER '16, 2016. [Online]. Available: https://doi.org/10.1109/SANER.2016.106

[45] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, ser. SP '20, 2020.

[46] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proc. of the 25th Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys.*, ser. ASPLOS '20, 2020, pp. 133–147. [Online]. Available: https://doi.org/10.1145/3373376.3378470

[47] E. Bauman, Z. Lin, and K. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. of the 25th Annual Network and Distributed System Security Symp.*, ser. NDSS '18, 2018.

[48] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[49] NSA, "Ghidra," https://ghidra-sre.org/, online; accessed 11 June 2020.

[50] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, ser. SP '16, 2016, pp. 138–157. [Online]. Available: https://doi.org/10.1109/SP.2016.17

[51] S. Alvarez, "Radare2," https://rada.re/n/, online; accessed 11 June 2020.

[52] P. Borrello, E. Coppa, and D. C. D'Elia, "Hiding in the particles: When return-oriented programming meets program obfuscation," ser. DSN'21, 2021, online extended version; https://arxiv.org/abs/2012.06658.

[53] M. Probst, A. Krall, and B. Scholz, "Register liveness analysis for optimizing dynamic binary translation," in *Proc. of the 9th Working Conf. on Rev. Engin.*, ser. WCRE '02, 2002, pp. 35–44. [Online]. Available: https://doi.org/10.1109/WCRE.2002.1173062

[54] D. C. D'Elia and C. Demetrescu, "On-stack replacement, distilled," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 166–180. [Online]. Available: https://doi.org/10.1145/3192366.3192396

[55] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[56] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012, pp. 319–328. [Online]. Available: https://doi.org/10.1145/2420950.2420997

[57] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed)," in *Proceedings of the 14th ACM ASIA Conference on Computer and Communications Security*, ser. ASIACCS '19, 2019, pp. 15–27. [Online]. Available: https://doi.org/10.1145/3321705.3329819

[58] Z. L. Nemeth, "Modern binary attacks and defences in the Windows environment – fighting against Microsoft EMET in seven rounds," in *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*, 2015, pp. 275–280.

[59] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. of the 23rd USENIX Security Symposium*, 2014, pp. 401–416.

[60] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou, "Detecting ROP with statistical learning of program characteristics," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17, 2017, pp. 219–226. [Online]. Available: https://doi.org/10.1145/3029806.3029812

[61] P. Larsen, S. Brunthaler, and M. Franz, "Automatic software diversity," *IEEE Sec. and Priv.*, no. 2, pp. 30–37, 2015. [Online]. Available: https://doi.org/10.1109/MSP.2015.23

[62] F. Biondi, S. Josse, A. Legay, and T. Sirvent, "Effectiveness of synthesis in concolic deobfuscation," *Computers & Security*, vol. 70, pp. 500–515, 2017. [Online]. Available: https://doi.org/10.1016/j.cose.2017.07.006

[63] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. of the Network and Distributed System Security Symposium*, ser. NDSS '08, 2008.

[64] C. S. Collberg and J. Nagra, *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection*, ser. Addison-Wesley Software Security Series. Addison-Wesley, 2010.

[65] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. and Trends in Prog. Lang.*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: http://dx.doi.org/10.1561/2500000014

[66] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86 - A platform for analyzing x86 executables," in *Proceedings of the 14th International Conference on Compiler Construction*, ser. CC '05, 2005, pp. 250–254. [Online]. Available: https://doi.org/10.1007/978-3-540-31985-6_19

[67] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution," in *2011 6th International Conference on Malicious and Unwanted Software*, 2011, pp. 58–65. [Online]. Available: https://doi.org/10.1109/MALWARE.2011.6112327

[68] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao, "Software watermarking using return-oriented programming," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '15. ACM, 2015, pp. 369–380. [Online]. Available: https://doi.org/10.1145/2714576.2714582

[69] K. Lu, S. Xiong, and D. Gao, "RopSteg: Program steganography with return oriented programming," in *Proc. of the 4th ACM Conf. on Data and App. Sec. and Priv.*, ser. CODASPY '14, 2014, pp. 265–272. [Online]. Available: https://doi.org/10.1145/2557547.2557572

[70] Christopher Domas, "M/o/Vfuscator," https://github.com/xoreaxeaxeax, 2015, online; accessed 11 June 2020.

[71] J. Kirsch, C. Jonischkeit, T. Kittel, A. Zarras, and C. Eckert, "Combating control flow linearization," in *Proceedings of the 32nd international conference on ICT Systems Security and Privacy Protection*, ser. IFIP SEC '17, vol. 502. Springer, 2017, pp. 385–398. [Online]. Available: https://doi.org/10.1007/978-3-319-58469-0_26

[72] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. of the 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, pp. 265–278. [Online]. Available: https://doi.org/10.1145/1950365.1950396

[73] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," in *Proc. of the 19th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 239–254. [Online]. Available: https://doi.org/10.1145/2541940.2541977

[74] I. Gouy, "The Computer Language Benchmarks Game," https://benchmarksgame-team.pages.debian.net/benchmarksgame/, 2018, online; accessed 20 March 2021.

[75] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu, "On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages," in *Proc. of the ACM Int. Conf. on Object Oriented Prog. Systems Lang. and Applications*, ser. OOPSLA '12, 2012, pp. 195–212. [Online]. Available: https://doi.org/10.1145/2384616.2384631

[76] D. C. D'Elia and C. Demetrescu, "Flexible On-stack Replacement in LLVM," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16, 2016, pp. 250–260. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854061

[77] B. Trower, "base64," http://base64.sourceforge.net/b64.c, 2001.

[78] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Memory models in symbolic execution: key ideas and new thoughts," *Soft. Testing, Verification and Reliability*, vol. 29, no. 8, 2019. [Online]. Available: https://doi.org/10.1002/stvr.1722