Evaluation of the Executional Power in Windows using Return Oriented Programming

Daniel Uroz
Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain
duroz@unizar.es

Ricardo J. Rodríguez*
Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain
rjrodriguez@unizar.es

***Abstract*—Code-reuse techniques have emerged as a way to defeat the control-flow defenses that prevent the injection and execution of new code, as they allow an adversary to hijack the control flow of a victim program without injected code. A well-known code-reuse attack technique is Return-Oriented-Programming (ROP), which considers and links together (relatively short) code snippets, named ROP gadgets, already present in the victim's memory address space through a controlled use of the stack values of the victim program. Although ROP attacks are known to be Turing-complete, there are still open question such as the quantification of the executional power of an adversary, which is determined by whatever code exists in the memory of a victim program, and whether an adversary can build a ROP chain, made up of ROP gadgets, for any kind of algorithm. To fill these gaps, in this paper we first define a virtual language, dubbed ROPLANG, that defines a set of operations (specifically, arithmetic, assignment, dereference, logical, and branching operations) which are mapped to ROP gadgets. We then use it to evaluate the executional power of an adversary in Windows 7 and Windows 10, in both 32- and 64-bit versions. In addition, we have developed ROP3, a tool that accepts a set of program files and a ROP chain described with our language and returns the code snippets that make up the ROP chain. Our results show that there are enough ROP gadgets to simulate any virtual operation and that branching operations are the less frequent ones. As expected, our results also indicate that the larger a program file is, the more likely to find ROP gadgets within it for every virtual operation.**

***Index Terms*—ROP chain, evaluation, Turing-completeness, Windows, automatic exploit**

I. INTRODUCTION

Software systems have increased in complexity and in size (measured as lines of code) during the last years. Nowadays, large software development teams are involved in several software projects at the same time, having a fixed time-to-market that urges them to end the development cycle as fast as possible, regardless of the software quality. Although automatic methods exist to improve the software quality, software vulnerabilities have dramatically increased, opening a window of opportunity to malicious exploitation [1].

*Corresponding author. This work was supported in part by MICINN under grant MEDRESE-RTI2018-098543-B-I00, by the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-17R), and by the University of Zaragoza and the *Fundación Ibercaja* under grant JIUZ-2020-TIC-08. The research of D. Uroz was also supported by the Government of Aragón under a DGA predoctoral grant (period 2019-2023).

Many of these vulnerabilities lead to control-hijacking attacks, which are the most popular category of memory exploits nowadays [2]. These attacks use code injection or its evolution, code-reuse attacks, to hijack the legitimate control flow of a victim program and execute malicious code. As a consequence, several defense approaches for control-hijacking attacks have been proposed [3], aiming to guarantee that the control flow of a program legitimately prevails. Examples of these include the use of stack cookies [4], inline software guards [5], runtime elimination of memory errors, control-flow integrity (CFI) [6]–[8], protection of data and code pointers [9], address space layout randomization (ASLR) [10], and write-xor-execute (W⊕X; also known as data-execution prevention, DEP) [11].

Code-reuse techniques have emerged as a trend of advanced threats to mitigate the effects of the control-flow defenses that prevent the injection and execution of new code. These techniques allow an adversary to hijack the control flow of a victim program to perform malicious activities without injected code.

Return-Oriented-Programming (ROP) is a code-reuse attack technique presented in 2007 for the x86 architecture (as an evolution of the *return-to-libc* attacks) [12], [13]. ROP attacks have been demonstrated feasible in numerous architectures, such as RISC [14], Linux/86 and Solaris/SPARC architectures [3], and recently in RISC-V [15]. In particular, ROP considers and links together (relatively short) code snippets already present in the process's memory address space, named as *ROP gadgets*. Each code snippet ends with an instruction that changes the program control flow (e.g., a `ret` instruction in Intel architectures), thus allowing an attacker who controls the stack to *chain* them together, controlling the order of code execution through the stack values. A chain of ROP gadgets is normally termed as a *ROP chain*. As the ROP chain links together these code snippets stored in memory pages marked as executable, ROP is able to evade control-flow defenses such as W⊕X.

ROP attacks are defeated with other control-flow defenses such as CFI. The security analysis of CFI solutions was first carried out in [16], raising questions on the true effectiveness of these solutions. The recent work in [17] presents a solution to precisely measure and verify the effectiveness of existing CFI solutions.

Modern operating systems such as Windows 10 incorporate

native defense techniques based on CFI, as Control-Flow Guard (CFG) [18]. CFG prevents the exploitation of memory corruption vulnerabilities, ensuring that the control-flow of the program remains legitimate. This defense is implemented in kernel-space and at program execution the targets of indirect branches are checked to verify whether they are valid targets. CFG, though, is not system-wide as it only works with "CFG-aware" programs, i.e., programs that are compiled with this feature enabled. Hence, CFG requires support from the compiler and the operating system to fully implement it.

Unfortunately, not many Windows programs incorporate this feature at the moment of this writing. To assess this claim, we have downloaded the top 10 of Windows applications in FossHub, a popular website hub for free and open-source software downloads[1]. and installed on a Windows 10 Enterprise (build 17763.rs5 release 18914-1434). From a total of 1034 program files (shared libraries and executable files), we found only 21 (which counts for 2.03%) CFG-aware programs. In contrast, 90.45% (4167 files) of the program files in the Windows system folder are CFG-aware programs. In any event, the hijacking of the control flow via return address corruption (as ROP does) is a known limitation of CFG. This hijacking technique is avoided with the shadow stack mechanism [19], a new security defense incorporated in the last versions of Windows that require both hardware and compiler support too.

A natural question that arises in this context is to analyze the capabilities of an adversary, regardless of the control-flow defenses put in place. ROP attacks are known to be Turing-complete [13], i.e., *ROP attacks are capable of any arbitrary computation*. Likewise, enough ROP gadgets to make up a Turing-complete set of operations have been found in Linux environments [20]. However, a question that still remains open is the quantification of the *executional power* of an adversary, which is determined by whatever code exists in the memory of a victim program. Noting that the ROP chain built by an adversary can be seen as the implementation of an algorithm designed to perform a desired task, if an adversary can find enough ROP gadgets for any arbitrary operation thus any algorithm can be implemented with a ROP attack. The more existing code is in a victim program, the more likelihood there is for finding useful gadgets [12]. Likewise, if the adversary is unable to find a ROP gadget for a specific operation needed in the ROP chain, the attack will likely fail.

Formally speaking, any real world computation can be translated into an equivalent computation that involves a Turing machine under the Church-Turing thesis [21]. Assuming this thesis holds, we can build a Turing machine that performs equivalent computations to the operations performed by a ROP chain.

To this extent, in this paper we first define a Turing-complete set of operations that make up a virtual language. This virtual language, dubbed ROPLANG, defines a set of operations that are later mapped to specific ROP gadgets, thus representing a ROP chain in an abstract way. We then use these operations to quantify the executional power of an adversary in a given environment. In particular, we evaluate the executional power of an attacker in Windows 7 and Windows 10, in its x86 and x86-64 versions, as Windows is still the predominant platform targeted by attackers [22].

As a side product of our research, we have developed a tool, dubbed `ROP3`, that accepts as input a set of program files and a ROP chain described with ROPLANG, and returns the ROP gadgets that make up the ROP chain. Our tool is built on top of the Capstone disassembly framework [23] and is designed to work with the Windows binary file format. In addition, our tool was designed in a modular way to facilitate the adoption of other file formats and to extend the virtual language.

Ethical considerations. The tool that we introduce in this paper can be used by an adversary to automatically generate a ROP chain and thus facilitate the exploitation of a victim program. However, at the same time this tool can be integrated in analysis workflows of automatic exploit generation [24], hence providing developers with enough information to prioritize the fixing of software bugs and the application of these fixes by end users.

This paper is organized as follows. Section II states the adversary model and the research questions. The related work is reviewed in Section III. Section IV describes the virtual language operations and proves its Turing-completeness, simulating a classical Turing machine. The evaluation of the executional power of adversaries in different flavors of Windows OS is given in Section V. Finally, Section VI concludes the paper and states future work.

II. Problem Definition

A. Adversary Model

We summarize in this section the number of assumptions made about the target system and the victim program in order to make the problem sufficiently tractable to allow for advancing the state-of-the-art. The assumptions are as follows:

- ASLR is not deployed on the target system, or there is a break available for ASLR. For instance, there exists a memory leak in the victim program that facilitates the adversary to break ASLR by means of learning function pointers of interest.
- CFI protection mechanisms are disabled on the victim program, or there is a break for CFI protection mechanisms which are put in place. For instance, a common bypass method in Windows CFG is leveraging the start of the ROP chain to non-CFG-aware images.
- The memory address space of the victim program is known to the adversary. For instance, the adversary can launch system commands and other analysis tools to know the dependencies of the victim program and the mapping of shared libraries. In conjunction with the first assumption, the adversary can look for and locate any byte within the memory address space of the victim program.

[1]The list of top 10 software downloaded from https://www.fosshub.com (accessed on December 24, 2020) is given in Appendix A.

B. Research Questions

In this paper, we aim to answer the following questions about ROP attacks:

RQ1. How often do ROP gadgets for any arbitrary operation arise in real-world programs?

RQ2. Is it possible to chain gadgets for a desired computation? Can adversaries build any kind of algorithm using a ROP chain?

III. Related Work

In this section, we focus on the most closely related work. In particular, we distinguish between works about ROP gadgets computation and works about specific ROP tools.

The seminal work of Shacham [12] showed that there are sufficient instructions in a piece of Intel x86 binary code large enough to make a ROP attack in a victim program viable. This is mainly motivated because the large instruction set architecture (ISA) of Intel x86 and because x86 instructions are unaligned in memory, and thus different instructions can be interpreted when an original instruction is read some bytes ahead. A taxonomy of ROP exploits with regards to the section code from which a ROP gadget is taken is given in [25].

Recall that a ROP attack chains code snippets ending in branch instructions that alter the control-flow of a victim program. Normally, these branch instructions refer to the `ret` instruction of Intel's ISA. As shown in [26], ROP attacks are feasible even without making use of these types of branch instructions.

A lot of tools have been proposed to detect and mitigate ROP attacks during the last decade. In the following, we discuss them in chronological order. First works were based in dynamic binary instrumentation (DBI). In [27], a DBI tool developed with the Pin framework [28] that uses taint analysis as an analysis technique for mitigating ROP attacks is introduced. Similarly, `DROP` is proposed in [29], which is a DBI tool developed with the Valgrind framework [30] and assumes that ROP gadgets contain no more than five instructions, ending in a `ret` instruction. Finally, in [31] the authors proposed `ROPDefender`, another tool based on Pin that detects ROP by means of a shadow stack. A similar approach based on shadow stacking is also proposed in [32], but residing in kernel-space instead.

`/ROP` is proposed in [33], which maintains an allow-listing of legitimate return addresses as a mitigation technique. `ROPGuard` [34] proposes a dynamic protection based on the runtime monitoring of Windows functions which are normally abused by ROP attacks. `kBouncer` [35] detects ROP exploits using hardware features of the Intel processors. In particular, it uses the Last Branch Recording registers, which is a hardware feature introduced in Intel Nehalem architectures that stores the most recent branches taken by the CPU as a performance optimization mechanism. Finally, a defense technique (for Linux i386 and AMD64 systems) named *disjoint code layouts* is introduced in [36], which relies on multiple executions of the same program under a control monitor component that guarantees that no code segments overlap. Other authors have proposed the use of hardware performance counters to detect ROP attacks at runtime [37].

Other authors have proposed tools more focused on offensive technology. `ROPInjector`, proposed in [38], transforms a shellcode to its ROP-chain equivalent and patches the program file to incorporate the ROP chain into its binary code directly. This way of action is uncommon for ROP attacking. In the same way, the authors in [39] proposed a metamorphic obfuscator dubbed `Frankenstein`, which is able to reassemble malicious code with code fragments entirely from other programs. In contrast, other authors have proposed the use of ROP as a defense technology. For instance, the authors in [40] use ROP it for software watermarking. In [41], a tool called `ROPOB` is proposed to obfuscate control flow using ROP. Likewise, `RopSteg` [42] hinders parts of the original program code by generating equivalent ROP gadgets and blending them into the program. The work in [43] proposes `SpecROP`, which mixes ROP attacks with speculative execution attacks. This novel speculative execution attack uses branch poisoning, commonly used in speculative execution attacks, to effectively stitch the execution of smaller gadgets equivalent to much larger, monolithic gadgets.

Regarding the generation and analysis of ROP chains, `deROP` [44] makes use of debugging functions to transform a ROP chain in a non-ROP chain for posterior analysis. In [45], a system that automatically generates ROP chains for Linux systems is introduced, although the described language is not Turing-complete, unlike ours. The work in [46] introduces Sigreturn Oriented Programming (SROP) as another evolution of ROP attacks. SROP abuses how signal returns are handled in Linux and other UNIX-based systems. The authors also showed that SROP is Turing-complete. `ROPEMU`, introduced in [47], is an emulation-based framework to analyze complex code reuse attacks, allowing for dissecting, reconstructing, and simplifying ROP chains.

In [48], the authors introduced a tool, named `AMOCO`, which builds a directed acyclic graph and uses symbolic execution analysis [49] to automatically generate ROP chains. `AMOCO` avoids the use of an intermediate language to minimize execution time and errors when constructing the ROP chains. In contrast, our tool `ROP3` uses a tree-like structure and analyzes it with a backtracking algorithm. Moreover, it relies on ROPLang, which can be seen as a sort of intermediate language. The ease of adding new virtual operations in ROPLang facilitates to find ROP gadgets that are semantically equivalent to an operation of interest.

Other tools widely used for building ROP chains are `ropper` [50], `ROPgadget` [51], and `ropium` [52], which also rely on the Capstone engine. Unlike our tool, `ropper` [50] does not automatically generate ROP chains. Although it has recently incorporated a semantic search for ROP gadgets, it is not backing in any formal language as we do with ROPLang. `ROPgadget` [51] provides a way to automatically get a Python code implementing a ROP chain that will run a shell. The definition of common shellcodes

Table I
SIMULATION OF ARITHMETIC OPERATIONS. THE **RET** INSTRUCTION (AT THE END OF EACH ROP GADGET) WAS DELIBERATELY OMITTED.

Operation	ROP gadgets/Operations
add(dst, src)	**add** dst, src
	clc **adc** dst, src
	inc dst
sub(dst, src)	**sub** dst, src
	clc **sbb** dst, src
	dec dst
neg(dst)	**xor** REG1, REG1 **sub** REG1, dst mov(dst, REG1)
	neg dst

Table II
SIMULATION OF ASSIGNMENT OPERATIONS. THE **RET** INSTRUCTION (AT THE END OF EACH ROP GADGET) WAS DELIBERATELY OMITTED.

Operation	ROP gadgets
mov(dst, src)	**mov** dst, src
	xchg dst, src
	xor dst, dst **add** dst, src
	xor dst, dst **not** dst **and** dst, src
	clc **cmovnc** dst, src
	stc **cmovc** dst, src
	push src **pop** dst
lc(dst, *value*)	**pop** dst*; value is set in the stack*
	popad*; value is set in the stack appropriately*

Table III
SIMULATION OF DEREFERENCE OPERATIONS. THE **RET** INSTRUCTION (AT THE END OF EACH ROP GADGET) WAS DELIBERATELY OMITTED.

Operation	ROP gadgets
ld(dst, src)	**mov** dst, [src]
st(dst, src)	**mov** [dst], src

using ROPLANG is a very interesting idea that needs further research. Finally, ropium [52] (built on top of ROPgadget) allows for searching for ROP gadgets via semantic queries too, but without relying on any formal language either.

Last, but not least, the work in [53] studied how software diversification can eliminate almost all the ROP gadgets in real-world applications. The language and the tool that we propose in this paper can be used in this context, helping software developers deploy their software with as few ROP gadgets as possible.

IV. DEFINITION OF THE VIRTUAL LANGUAGE: ROPLANG

A. Virtual Language Operations

A virtual operation in ROPLANG is simulated using a concrete x86 instruction sequence in the vulnerable program execution that makes up ROP gadgets. Our operations use a similar notation to Intel's assembly notation, which is also used in the x86 instructions shown in the rest of this section. In addition, we adhere to the Intel x86 syntax when having operations with two operands (the destination register first, and the source register second). Such instruction sequences or gadgets can be divided into the following categories:

1) Arithmetic operations: The basic arithmetic operations that we considered are *addition* (add), *subtraction* (sub), and *negation* (neg), which can be simulated using a variety of x86 instructions sequences: For instance, to simulate an addition we can rely on the own assembly instruction **add** operation, on the instruction **inc**, or on the combination of **clc** and **adc** instructions. Table I shows examples of x86 ROP gadgets that simulate add, sub, and neg.

In the case of having a ROP gadget with more than one assembly instruction, it could be divided into smaller ROP gadgets, where every gadget is composed of few instructions. For instance, the **clc;adc** dst, src can be simulated either by a single ROP gadget having both instructions and ending with a return instruction, or by two ROP gadgets, **clc** and **adc** dst, src, ending both with a return instruction as **ret** does not affect any flag when executed.

The virtual operations of ROPLANG can have relationships between them too. For instance, one of the ways to simulate the neg operation (see the first ROP gadget of neg in Table I) has as last instruction a *move* (mov) operation, which is defined next. In addition, this simulated operation also defines REG1 as an abstract representation of any general-purpose register different from dst.

2) Assignment operations: The assignment operations enable us to assign value to a variable. A variable in this context is a CPU logical (also called *general-purpose*) register, while a value can be either an immediate or a logical register too. In this category, we have considered the *move* (mov) and *load constant* (lc) operations. Examples of ROP gadgets simulating these operations are shown in Table II. Note that some instructions may overwrite certain general-purpose registers during its execution. These registers are normally called *clobbered registers*. We say that a ROP gadget of an operation can have *side effects* if the execution of the gadget reads/writes memory addresses or overwrites more clobbered registers than the minimum needed for the operation. For instance, the **popad** instruction, used to simulate the lc operation (see Table II), will load double-word values (32-bit length) into all the general-purpose registers, apart from the destination register of interest.

3) Dereference operations: The *load* (ld) and *store* (st) operations represent memory dereferences, which are useful to visit a memory location for reading or writing. Table III shows examples of these operations.

4) Logical operations: This category includes the xor, and, or, and not operations. Note that by De Morgan's Laws, the logic operations can be simplified to an operation {and, or} plus an operation of the set {xor, not, neg}. Straightforward examples of ROP gadgets simulating these operations are presented in Table IV.

Table IV
SIMULATION OF LOGICAL OPERATIONS. THE **RET** INSTRUCTION (AT THE END OF EACH ROP GADGET) WAS DELIBERATELY OMITTED.

Operation	ROP gadgets
xor(dst, src)	**xor** dst, src
and(dst, src)	**and** dst, src
or(dst, src)	**or** dst, src
not(dst)	**not** dst **xor** dst, 0xFFFFFFFF

5) Branching operations: Conditional branching operations require some tricky steps to be achieved. In particular, we need to stitch together various ROP gadgets to obtain an operation equivalent to conditional branching. We need first to undertake some operation to perform the desired comparison. As stated in [12], the carry flag (CF) is enough to obtain the full set of standard comparisons. As *comparison* operations, we define the *equal comparison* (eqc), and the *less than comparison* (ltc). Note that having eqc and ltc we can implement other comparison operations such as greater than or equal, distinct, etc. For instance, to perform a distinct comparison we can perform first an equal comparison and then negate the result. All the comparison operations will clear (or set) the CF according to the comparison performed. Table V shows a subset of ROP gadgets useful to simulate comparison operations.

Once the comparison has been done, we can make the conditionally change in the stack pointer depending on the result of the comparison. Following the method proposed in [12], we define the following operations:

1) First, we need then to get the CF in a general-purpose register. In this regard, we can use assembly instructions that work explicitly with the CF, such as left/right rotations with carry (**rcl**, **rcr**), or addition with carry (**adc**). Other instructions such as subtraction with carry (**sbb**) can also be used, although additional instructions will be needed in this case (see Table VI). We named this operation as *get carry flag* operation (gcf). This operation needs a comparison operation as a parameter.
2) When the gcf operation is done, we have a general-purpose register that contains a 1 or 0 value. As proposed in [12], we transform it to contain an arbitrary δ value or 0, where δ represents the offset that we want to add to the stack pointer register if the condition checked in the first step holds. We define the *load stack delta* operation (lsd) that enables us to set either δ or 0 in a general-purpose register.
3) Finally, we modify the value of the stack pointer register appropriately. We define a *stack pointer addition* (spa) that performs an addition to the stack pointer register of any other general-purpose register. If the register contains a negative value, then the addition becomes a subtraction operation. Note that we can also use a spa operation preceded by a neg operation to simulate a stack pointer subtraction. For the sake of completeness, we also define a *stack pointer subtraction* (sps) operation that subtracts to the stack pointer register the value of any other general-purpose register.

Table V
SIMULATION OF COMPARISON OPERATIONS.

Operation	Operation
eqc(dst, src)	sub(dst, src) neg(dst)
ltc(dst, src)	sub(dst, src)

Table VI
SIMULATION OF CONDITIONAL BRANCHING OPERATIONS. THE **RET** INSTRUCTION (AT THE END OF EACH ROP GADGET) WAS DELIBERATELY OMITTED.

Operation	ROP gadgets/Operations
gcf(dst$_{CF}$, cop(dst, src))	lc(REG1, 0) *Comparison operation* cop(dst, src) **adc** dst$_{CF}$, REG1
	lc(REG1, 0) *Comparison operation* cop(dst, src) **sbb** dst$_{CF}$, REG1 neg(dst$_{CF}$)
	lc(dst$_{CF}$, 0) *Comparison operation* cop(dst, src) **rcl** dst$_{CF}$, 1
lsd(dst$_{CF}$, δ)	lc(REG1, δ) neg(dst$_{CF}$) and(dst$_{CF}$, REG1)
spa(src)	add(REG_SP, src)
sps(src)	sub(REG_SP, src)

Regarding unconditional branching, a virtual operation can be straightforwardly constructed relying on a similar way to conditional branching. We define the jmp operation such that it makes use of lc to load a δ offset in a register, and then uses spa to unconditionally change the control-flow of the ROP chain. This operation is shown in Table VII.

Tables I to VII show a (non-exhaustive) list of the ROP gadgets needed to simulate the virtual operations defined by our language. Note that some operations are defined with other virtual operations too. We assume that no harmful side effects (i.e., the execution of ROP gadgets do not affect to posterior computations within the same operation) are produced between these sequences of virtual operations.

B. On the Turing-Completeness of ROPLANG

Following a similar approach as in [54], we show how a classical Turing machine can be simulated with ROPLANG to demonstrate that it is Turing-complete. Formally speaking, a (one-tape) Turing machine $\mathcal{M}$ is a tuple $\langle Q, \Gamma, \sigma_0, \Sigma, q_0, F, \delta \rangle$ where [55]:

- Q is a finite, non-empty set of states;
- Γ is the finite set of symbols in the tape alphabet;

Table VII
SIMULATION OF UNCONDITIONAL BRANCHING OPERATIONS.

Operation	ROP gadgets/Operations
jmp(dst, δ)	lc(dst, δ) spa(dst)

- $\sigma_0 \in \Gamma$ is the blank symbol;
- $\Sigma \subseteq \Gamma \setminus \{\sigma_0\}$ is the input alphabet symbols, i.e., the set of symbols allowed to appear in the initial tape contents;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states or accepting states. If any state of F is reached, the input string (initial tape contents) is accepted; and
- $\delta : (Q \setminus F) \times \Gamma \rightarrow (\Gamma \times \{L, R\} \times Q)$ is a partial function called transition function which determines the next move, where L is left shift and R is right shift.

The mechanics of a Turing machine $\mathcal{M}$ is defined by δ. If $\delta(q_i, S_j) = (S_{i,j}, D, q_{i,j})$, then when state of the machine is q_i, reading the symbol S_j on the tape makes $\mathcal{M}$ to replace S_j by $S_{i,j}$, moving the tape in direction $D \in \{L, R\}$ and going to state $q_{i,j}$. If δ is not defined on the current state and the current tape symbol, then the machine halts. If $q_{i,j} \in F$, then it is said that $\mathcal{M}$ stops and accepts the initial tape contents.

Roughly speaking, the operation of the Turing machine comprises four steps. First, it reads the current tape symbol. The current tape symbol and the current state are then used to check the transition table and get the next state and symbol. Next, the new symbol is written to the tape and the current state is updated appropriately. Finally, the tape head is moved to the left or right direction.

Representation. To represent a Turing machine, we set-up the following data structures in the victim program's memory: $q_{cur} \in Q$ holds the current state, t_{head} tracks the position on the tape containing the current symbol $\sigma \in \Sigma$. As the tape is linear, a left shift movement on the tape with regard to the current position means $t_{head} - 1$, while $t_{head} + 1$ represents a right shift movement instead. The transition table t_{table} will be placed in memory such that the comparison operation to search is transformed into a direct lookup in a two-dimensional array.

Initialization. We use three general-purpose registers to internally store the references to t_{head}, q_{cur}, and t_{table} (`REG1`, `REG2`, and `REG3`, respectively). An adversary can craft a payload as shown in Listing 1 to load the input, the initial state, and the transition table. Additionally, we use a special symbol (♣) to encode the new symbol in final states for the sake of simplicity. We use another general-purpose register, `REG4`, to store it.

Listing 1. Initialization of a Turing machine.

```
1  mov(REG1, t_head) ; start loading tape content
2  st(REG1, S_0)
3  add(REG1, 1)
4  st(REG1, S_1)
5  ...
6  add(REG1, 1)
7  st(REG1, S_N) ; end loading tape content
8  mov(REG1, t_head) ; reset tape header ptr
9  mov(REG2, q_cur) ; initial state
10 mov(REG3, t_table) ; start loading trans. table
11 st(REG3, S_0,0) ; new symbol (q_0, S_0)
12 add(REG3, 1)
13 st(REG3, D_0,0) ; direction
14 add(REG3, 1)
15 st(REG3, q_0,0) ; new state
16 add(REG3, 1)
17 st(REG3, S_0,1) ; new symbol (q_1, S_0)
18 add(REG3, 1)
19 st(REG3, D_0,1) ; direction
20 add(REG3, 1)
21 st(REG3, q_0,1) ; new state
22 add(REG3, 1)
23 ...
24 st(REG3, S_0,i) ; new symbol (q_i, S_0)
25 add(REG3, 1)
26 st(REG3, D_0,i) ; direction
27 add(REG3, 1)
28 st(REG3, q_0,i) ; new state
29 add(REG3, 1)
30 ...
31 st(REG3, S_j,i) ; new symbol (q_j, S_i)
32 add(REG3, 1)
33 st(REG3, D_j,i) ; direction
34 add(REG3, 1)
35 st(REG3, q_j,i) ; new state
36 add(REG3, 1)
37 mov(REG3, t_table) ; reset trans. table ptr
38 mov(REG4, ♣) ; halt symbol
```

Simulating the mechanics of a Turing machine. As mentioned above, each step in the machine checks the transition table by using the current state and the current tape symbol. By the way used to represent the transition table, this checking procedure will be a direct lookup in a two-dimensional array. We use the `add` and `ld` operations to dynamically calculate the offset in the transition table, based on the current symbol and state. Once the correct offset is obtained, we use it to lookup the new symbol (writing it into the tape), the next state, and the direction to move the tape header. This process is repeated until the machine reaches a halt (or final) state. We use an operation equivalent to "no operation" to indicate the termination of the Turing machine. The payload needed to simulate the steps of a Turing machine is shown in Listing 2.

Listing 2. Simulating a step of a Turing machine.

```
1  _step:
2     mov(REG5, REG3) ; set to trans. table
3     add(REG5, REG1) ; get the row
4     ld(REG5, REG5)
5     add(REG5, REG2) ; get the column
6     ld(REG6, REG5) ; load new symbol
7     gcf(REG7, eqc(REG4, REG5)) ; halt sym?
8     lsd(REG7, _exit) ; finish simulation
9     spa(REG7)
10    st(REG1, REG6) ; write it to tape
11    add(REG5, 1)
12    ld(REG6, REG5) ; load direction
13    add(REG1, REG6) ; move the tape header
14    add(REG5, 1)
15    ld(REG2, REG5) ; load new state
16    jmp(REG5, _step) ; repeat the step
17 _exit:
18    mov(REG1, REG1) ; halt state
```

C. Tool Description

The `ROP3` tool is developed in Python programming language, and it relies on the Capstone disassembly framework [23] to search for gadgets, operations, and ROP chains.

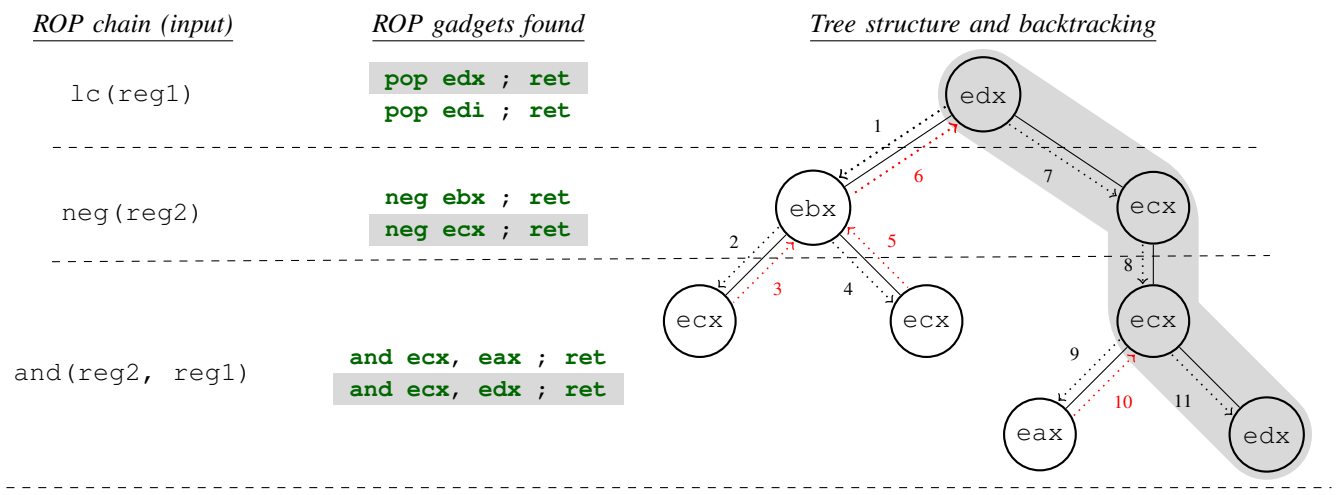

Figure 1. Backtracking algorithm of ROP3 to find a ROP chain.

Thanks to Capstone, the disassembled instruction counts on a very fine-grained level of detail, such as the list of implicit registers read/written, thus enabling the tool to make decisions based on the instruction and operand types.

Each of the virtual operations that make up ROPLANG (described in Section IV) is natively supported by ROP3. Specifically, a virtual operation is defined in a separated file using YAML syntax. An *operation* is composed of one or more *sets of instructions*, whereas a single *instruction* is defined by its *mnemonic* and the *operands* (if any). The operands can be defined to hold a particular logical register (such as **eax**, **rbx**, **r14**, etc.) or an arbitrary register (reg1, reg2, ...), which acts itself as a register mask (that is, it can match to any logical register). Optionally, an operand can hold an *arbitrary value* to comply semantically with an operation (e.g, the operand can hold -1 to obtain the negation of an integral value of a register using the xor operation). Likewise, an operand can be defined as *implicit* when the assembly instructions defined in the YAML file uses some registers implicitly. For instance, the **leave** instruction in Intel x86 assembly has no operands, but it implicitly moves the current value held on **ebp** to the **esp** register and pops the top of the stack into **ebp**. Following this structure, we can define an arbitrary number of custom operations, either as a single or as multiple YAML files.

ROP3 follows a similar approach as in [12] to search for ROP gadgets. It first locates the raw bytes of instruction opcodes that change the control flow in the executable sections of the program file (or files) provided as input, and then goes backwards at the defined maximum of byte length from such an instruction. The maximum size of the ROP gadgets can be parametrized when executing the tool and its default value is set to 5. ROP3 supports the search of different gadget endings, such as near returns (**retn**), far returns (**retf**), unconditional jumps (**jmp**), or procedure calls (**call**).

On a higher abstraction level, in ROP3 a *ROP chain* is the concatenation of a series of operations. Usually, there are data dependencies between the registers used by the operations in a ROP chain. For example, a ROP chain may firstly load a constant value in any logical register (operation lc(reg1) with ROPLANG syntax) and, afterwards, move such a value to another register (operation mov(reg2, reg1)). As ROPLANG, ROP3 follows the Intel syntax when referring to operands (that is, the first operand is the destination register and the second operand is the source register).

In order to search for ROP chains, ROP3 works as follows. The tool first parses the plain text file provided by the user as input, in which the operations to be chained are defined with ROPLANG operations. It then finds all gadgets that fulfill every operation, providing the destination and source registers of each operation and builds a tree structure, considering the operation order defined in the input file. Finally, it solves the data dependencies between operations traversing the tree recursively in depth-first order with backtracking.

Figure 1 illustrates how ROP3 works. Let us consider that the user wants to build a ROP chain defined by the operations lc(reg1); neg(reg2); and(reg2, reg1) (shown at the left side of the figure). ROP3 searches for ROP gadgets that fulfill every operation (obtaining the ones shown at the middle of the figure) and then builds a tree structure for every gadget found for the first operation. For the sake of illustrative purposes, we only show the tree of the first ROP gadget found (depicted at the right side of the figure). For every operand in an instruction, a node is created. Every node of the tree has been labeled with the register considered in each ROP gadget.

Once a tree is built, it is traversed with backtracking, solving the data dependencies between the operations as follows: the edx register is assigned to the register mask reg1 and then moves (denoted as a dotted arrow) to the node representing the first gadget of the second operation, **neg ebx ; ret** (1). In this node, the ebx register is assigned to the register mask reg2, and then moves to the next node (2), which represents **and ecx, eax ; ret**. As the assigned value to reg2 does not match with the current operand (registers **ecx**), it moves

up (3), denoted as a dotted arrow in red, and down (4) to the next child of the parent node. The use of backtracking allows ROP3 to prune the sub-trees and find a solution in an efficient manner. As the operand of the next node does not match either, it moves up again (5), going back to the previous node as no more nodes are available (6), and clears up the value assigned to reg2. Moving to the next gadget, **neg ecx ; ret** (7), it assigns **ecx** to reg2. It moves down (8) and checks that the assigned value of reg2 matches the value of reg2 in the gadget, and then moves down to check the second operand (9). As the assigned value of reg1 does not match the value of reg1 in this gadget, it moves up again (10) and goes to the next node (11). In this case, the assigned value of reg1 matches the value of reg1 and thus the search has finished. The search algorithm then backtracks till the root node, keeping track of all assigned register values, and returns the result to the user. The combination of ROP gadgets that satisfy the ROP chain has been highlighted in gray in the tree.

At the moment of this writing, we are adding to ROP3 a feature to consider the (possible) side effects of a sequence of instructions within a ROP gadget (for instance, an unwanted load of a value into a register, an addition between registers, etc.). For each instruction within a ROP gadget, we keep track of what is producing into the logical registers, thus pruning the sub-tree or continuing the path when appropriate.

For the sake of open science, we have released our software under the GNU/GPLv3 license [56]. In addition, ROP3 is also a Python3 library, which facilitates the integration with other analysis pipelines.

V. Evaluation

In this section, we first use ROP3 to evaluate how many ROP gadgets for any arbitrary operation arise in real-world programs. In particular, we focus on different flavors of Windows OS, checking whether the default dynamic link libraries (DLLs) shipped with Windows OS contain ROP gadgets for any arbitrary operation, as specified by ROPLANG. We then verify whether there are enough gadgets to build a fully operating Turing machine. Appendix B gives details to exercise our artifact and reproduce the evaluation.

A DLL is a Windows executable file that contains a set of functions and data used by other Windows programs. These functions are normally termed as Application Programming Interface (API) functions. They are similar to the .so files in Linux OS. A set of DLLs is shipped with Windows OS and is available for any Windows program, providing shared functions that make integration easier with the Windows OS itself. These DLLs are commonly referred to as *system DLLs*.

When the Windows PE loader loads a program for execution, all its shared libraries are loaded as well in the process' memory address space. Windows OS incorporates a mechanism to improve the application load time and to cache commonly used system DLLs. This mechanism, termed KnownDlls, is an object stored in the Windows Registry [57].

As the number of DLLs shipped with Windows OS is in terms of hundreds, being also different across the flavors of Windows, we have considered only the subset of system DLLs contained in KnownDlls that are common across all the versions of Windows considered for the experimentation.

This subset of DLLs is composed of a total of 20 DLLs, counting with DLLs frequently used by Windows programs. Apart from these ones, we have considered other DLLs such as msvcrt.dll (the core element of the Microsoft Visual C runtime library), psapi.dll (related to process management), and ws2_32.dll (related to Windows Sockets API), although they were not included into the KnownDlls object of certain Windows OSes considered in the experimentation. Likewise, we have also decided to include ntdll.dll in the selected subset, as it is always loaded with any Windows program file since it contains functionalities of the Windows PE loader itself [58]. We refer the reader to Figure 2 to see the complete set of DLLs considered for experimentation.

As a test-bed environment, we have made a fresh install of different Windows OSes in a set of virtual machines (having each a dynamically allocated hard-drive of 32GB and 4GiB of RAM memory) over the Oracle VirtualBox hypervisor. The installation process followed a default (out-of-the-box) configuration. Once installed, we have empirically verified the existence of the above subset of system DLLs and extracted them out of the virtual machine.

We tested the two most widely used versions of Windows OS (Windows 10, 75.68%, and Windows 7, 18.03%; numbers of the year 2020 according to [59]), grouped by architecture, are: [**32 bits**] *Windows 7 Professional 6.1.7601 Service Pack 1 Build 7601* and *Windows 10 Education 10.0.14393 Build 14393*; [**64 bits**] *Windows 7 Professional 6.1.7601 Service Pack 1 Build 7601* and *Windows 10 Pro 1703 Build 15063.726*. In the following, we refer to each version by the major version name and its architecture word size.

With regard to the configuration of ROP3, we configured it to search for gadgets of 10-byte length. Although the tool supports searching for different ROP gadget endings, we have only considered near returns because the other endings introduce more complexity when building a ROP chain. Additionally, we have counted only once the ROP gadgets composed of the same sequence of instructions, regardless of the memory addresses in which they are mapped.

We have searched the ROPLANG operations as defined in Section IV (that is, our results are biased by the current definition of operations). We have considered every ROP gadget composed of several instructions as a single gadget, instead of handling every instruction as an individual gadget. Furthermore, we have extended the spa operation to consider also the addition of immediate values such as 4, 8, 16, and 32 (spa-4, spa-8, spa-16, and spa-32 operations, respectively) to increase the probability of finding appropriate ROP gadgets. In the same way, we have divided the gcf operation into two, embedding into them the type of comparison being carried on: *equal comparison* (gcf-eqc) and *less-than comparison* (gcf-ltc) operations. In an attempt to

increase the likelihood of finding complex operations such as the branching operations, we have extended the `neg`, `eqc`, `gcf`, `lsd`, and `jmp` operations with the use of intermediate `mov` between their operations. Further discussion is given in Section V-B.

A. Prevalence of ROP Gadgets

For each operation, we have computed its percentage of occurrence within every DLL and plotted it in a heatmap. In addition, we have also annotated the number of results found by `ROP3`, setting only the most significant digit and the order of magnitude when the number of results are greater than 10^4 for the sake of readability. DLLs have been sorted by their size in each heatmap. Figure 2 shows the results for Windows 7 and Windows 10 in 32-bit (top figures) and 64-bit (bottom).

Our experiments show that the branching virtual operations are the less frequent operations, regardless of the architecture. Moreover, in 32 bits there are no results for any of the comparison operations as defined in Table VI, regardless of the Windows version. In fact, these operations only appear in one DLL of Windows 7 SP1 64-bit. The case of the unconditional branch operation is very interesting, as we have (few) results in 32-bit versions, while none in 64-bit.

For the rest of virtual operations, the results are diverse, although we can find at least one result for each virtual operation. The results tend to be higher in the lower part of the figures, which corresponds with the bigger DLLs. As claimed by other authors [12], the longer the binary code is, the more likely to find ROP gadgets to perform any arbitrary operation.

Regarding the versions of Windows, the results show the number of virtual operations found in Windows 10 is always greater than in Windows 7. This can be motivated because of the difference in sizes between the DLLs, as in Windows 10 the DLLs are normally bigger than in Windows 7. Likewise, we have empirically verified that the size in 64-bit is always greater than in 32-bit (but in `msvcrt.dll`) for both versions of Windows, which also explains the slight variations in the results among the architecture word sizes.

It is worth mentioning also that the 64-bit mode in Intel introduced a new addressing form named relative Instruction Pointer addressing (RIP-relative addressing) [60], which is the default for many 64-bit instructions that reference memory in any of their operands. Therefore, none of the 64-bits instructions contain absolute memory addresses. In contrast, 32-bit instructions within a DLL can contain references to memory addresses with regard to its base address, which is randomized in every Windows booting due to ASLR [58]. Therefore, unlike the results of 64-bit, the results shown in 32-bit strongly rely on the base addresses of the DLLs, and may change when the base addresses are different. Further experimentation is needed to evaluate how ASLR can affect the prevalence of ROP gadgets in 32-bit Windows systems.

B. Simulating a Turing machine

To simulate a Turing machine as defined previously, we would need to find at least one result for every virtual operation. However, the current definition of the virtual operations provides very limited results for conditional and unconditional operations in certain versions of Windows, which are mandatory to simulate a Turing machine.

Note that the virtual operations in Section IV-A are defined in their simplest form. For instance, the virtual operation `eqc(dst, src)` is defined as `sub(dst, src); neg(dst)` (see Table V). We can add an intermediate assignment operation (as `mov(reg1, dst)`) to relax the data dependency constraints as it is more likely to find a `mov(reg1, dst)` operation, although the length of the ROP chain would increase. The virtual operation `eqc(dst, src)` can then be defined as `sub(dst, src); mov(reg1, dst); neg(reg1)`. These modifications can be done to any virtual operation of ROPLANG, without affecting its semantics.

We have performed this extension (by extending the corresponding YAML files) to the definition of the `neg`, `eqc`, `gcf`, `lsd`, and `jmp` operations and recomputed the results, which are plotted in Figure 3. With this variation, there are a lot of results for conditional and unconditional operations in 32-bit, while the number of results in 64 bits are more discrete. The only operation that has still no results is the unconditional operation in Windows 7 SP1 64 bits. Therefore, these experimental results show us that an adversary can very likely find a ROP gadget for any arbitrary operation, just doing sophisticated linking of other operations when the operation needed is not directly found. As shown, our tool supports the extension of the virtual operations in a very straightforward way, increasing the likelihood of finding operations of interest.

VI. Conclusions and Future Work

Although ROP attacks are known to be Turing-complete, the executional power of an adversary strongly relies on the code which already exists in the memory of a victim program. In this paper, we have investigated the executional power by evaluating how often the ROP gadgets for any arbitrary operation arise in a subset of commonly used system shared libraries of Windows 7 and Windows 10, both in 32-bit and 64-bit architecture word sizes. In addition, we have also investigated if there are enough gadgets to be chained for any arbitrary computation. To do so, we have defined a virtual language, dubbed ROPLANG, whose operations are mapped to ROP gadgets. We also introduced a new tool, named `ROP3`, which allows for finding ROP gadgets and building a ROP chain specified by ROPLANG's operations in any set of program files given as input.

Our experimental results show that any virtual operation is found, being the branching operations the less frequent ones. Furthermore, the size of the program file clearly impacts on the prevalence of ROP gadgets within the file. We have also shown that a careful linking of the virtual operations can be performed to find operations that are not straightforwardly found.

As future work, we aim to improve `ROP3` to automatically eliminate the side-effects likely to occur by some ROP gadgets in a generated ROP chain. Furthermore, we aim to evaluate and to compare the executional powers in other operating systems such as UNIX-based systems and macOS, among others.

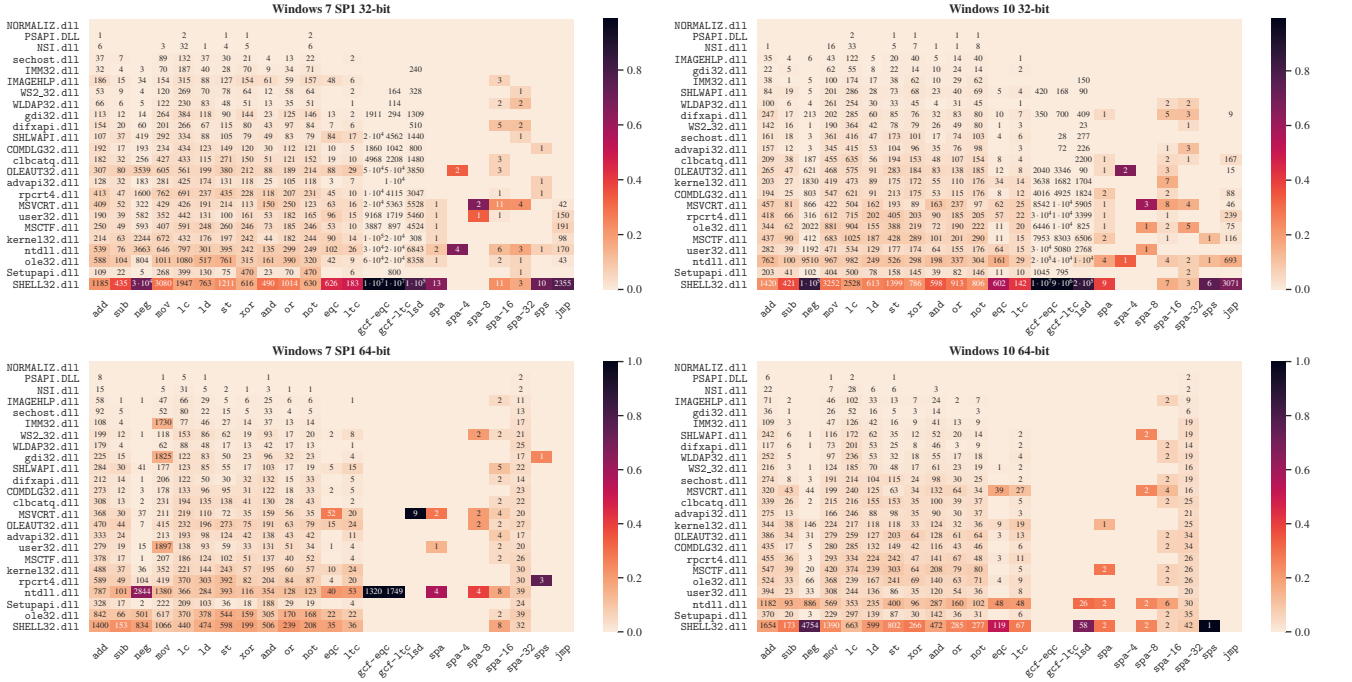

Figure 2. Number of ROP gadgets per ROPLANG's operation in Windows 7 and Windows 10 in 32-bit (top figures) and 64-bit (bottom figures).

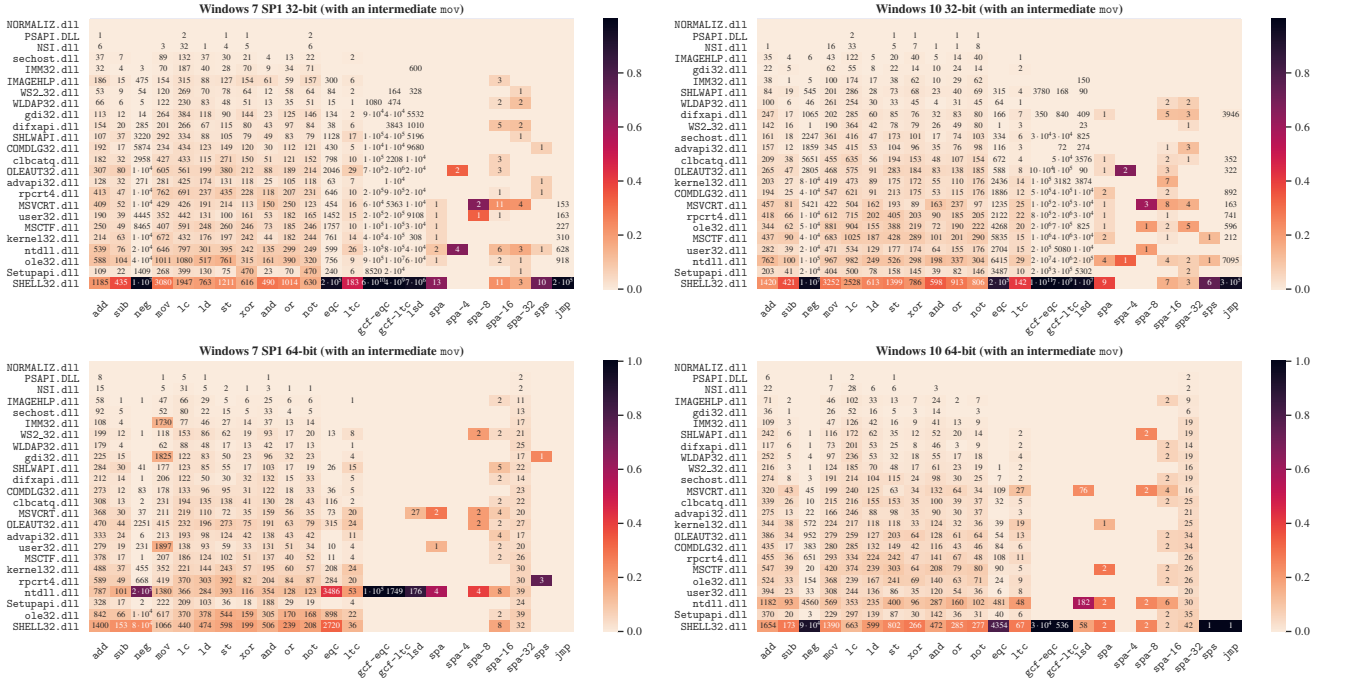

Figure 3. Number of ROP gadgets per ROPLANG's operation, considering intermediate `mov` operations in the `neg`, `eqc`, `gcf`, `lsd`, and `jmp` operations, in Windows 7 and Windows 10 in 32-bit (top figures) and 64-bit (bottom figures).

REFERENCES

[1] P. Johnson, D. Gorton, R. Lagerström, and M. Ekstedt, "Time between vulnerability disclosures: A measure of software product vulnerability," *Computers & Security*, vol. 62, pp. 278–295, 2016.

[2] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5.

[5] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2006, pp. 75–88.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009.

[7] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352.

[8] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[9] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163.

[10] PaX Team, "Address Space Layout Randomization (ASLR)," Online; http://pax.grsecurity.net/docs/aslr.txt., Mar. 2003, accessed on December 26, 2020.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity - Principles, Implementations, and Applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2005, pp. 340–353.

[12] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2007, pp. 552–561.

[13] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 121–141.

[14] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008, pp. 27–38.

[15] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-Oriented Programming on RISC-V," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 471–480.

[16] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 401–416.

[17] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1821–1835.

[18] Microsoft, "Control Flow Guard," Online; https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard., May 2018, accessed on December 26, 2020.

[19] H. Pulapaka, "Understanding Hardware-enforced Stack Protection ," Online; https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815., Mar. 2020, accessed on December 24, 2020.

[20] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA: USENIX, 2012.

[21] B. J. Copeland, "The Church-Turing Thesis," in *The Stanford Encyclopedia of Philosophy*, summer 2020 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2020, accessed on March 11, 2021.

[22] R. Benzmüller, "Malware trends 2017," [Online; https://www.gdatasoftware.com/blog/2017/04/29666-malware-trends-2017], Oct. 2017, accessed on October 04, 2017.

[23] Capstone, "Capstone – The Ultimate Disassembler," Online, http://www.capstone-engine.org/, accessed on July 10, 2020.

[24] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, February 2014.

[25] T. Müller, "ASLR Smack & Laugh Reference," RWTH Aachen, Germany, Seminar on Advanced Exploitation Techniques, February 2008.

[26] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2010, pp. 559–572.

[27] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks," in *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing (STC)*. New York, NY, USA: ACM, 2009, pp. 49–54.

[28] 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 (PLDI)*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[29] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting Return-Oriented Programming Malicious Code," in *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, ser. Lecture Notes in Computer Science, vol. 5905. Springer Berlin Heidelberg, 2009, pp. 163–177.

[30] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[31] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 40–51.

[32] T. Shuo, H. Yeping, and D. Baozeng, "Prevent Kernel Return-Oriented Programming Attacks Using Hardware Virtualization," in *Proceedings of the 8th International Conference on Information Security Practice and Experience (ISPEC)*, ser. Lecture Notes in Computer Science, M. Ryan, B. Smyth, and G. Wang, Eds., vol. 7232. Springer Berlin Heidelberg, 2012, pp. 289–300.

[33] J. DeMott, "Microsoft BlueHat Prize Submission," Tech. Rep., 2012.

[34] I. Fratric, "Runtime Prevention of Return-Oriented Programming Attacks," Microsoft BlueHat Prize Submission, Tech. Rep., 2012.

[35] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 447–462.

[36] S. Volckaert, B. Coppens, and B. D. Sutter, "Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution," *IEEE Trans. Dependable Secure Comput.*, vol. 13, no. 4, pp. 437–450, July 2016.

[37] S. Das, B. Chen, M. Chandramohan, Y. Liu, and W. Zhang, "ROPSentry: Runtime defense against ROP attacks using hardware performance counters," *Computers & Security*, vol. 73, pp. 374–388, 2018.

[38] C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis, "Transforming malicious code to ROP gadgets for antivirus evasion," *IET Information Security*, vol. 13, no. 6, pp. 570–578, 2019.

[39] V. Mohan and K. W. Hamlen, "Frankenstein: Stitching Malware from Benign Binaries," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, ser. WOOT'12. USA: USENIX Association, 2012, p. 8.

[40] 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. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 369–380.

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

[42] K. Lu, S. Xiong, and D. Gao, "RopSteg: Program Steganography with Return Oriented Programming," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 265–272. [Online]. Available: https://doi.org/10.1145/2557547.2557572

[43] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative Exploitation of ROP Chains," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 1–16.

[44] K. Lu, D. Zou, W. Wen, and D. Gao, "deRop: Removing Return-oriented Programming from Malware," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. New York, NY, USA: ACM, 2011, pp. 363–372.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 2011 USENIX Security Symposium*, 2011, pp. 1–16.

[46] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, May 2014, pp. 243–258.

[47] M. Graziano, D. Balzarotti, and A. Zidouemba, "ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 47–58.

[48] Y. Wei, S. Luo, J. Zhuge, J. Gao, E. Zheng, B. Li, and L. Pan, "ARG: Automatic ROP Chains Generation," *IEEE Access*, vol. 7, pp. 120 152–120 163, 2019.

[49] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, May 2018.

[50] S. Schirra, "Ropper version 1.13.6," [Online; https://scoding.de/ropper/], Feb. 2021, accessed on March 12, 2021.

[51] J. Salwan, "Ropgadget version 6.5," [Online; https://github.com/JonathanSalwan/ROPgadget], Jan. 2021, accessed on March 12, 2021.

[52] B. Milanov, "ROPium version 3.2," [Online; https://github.com/Boyan-MILANOV/ropium], Mar. 2020, accessed on March 12, 2021.

[53] J. Coffman, D. M. Kelly, C. C. Wellons, and A. S. Gearhart, "ROP Gadget Prevalence and Survival Under Compiler-based Binary Diversification Schemes," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. SPRO '16. New York, NY, USA: ACM, 2016, pp. 15–26.

[54] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986.

[55] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson Education, Inc., 2007.

[56] D. Uroz and R. J. Rodríguez, "rop3 version 0.9," [Online; https://github.com/reverseame/rop3], Mar. 2021, accessed on March 12, 2021.

[57] L. Osterman, "What are Known DLLs anyway?" [Online; available at https://blogs.msdn.microsoft.com/larryosterman/2004/07/19/what-are-known-dlls-anyway/], Jul. 2004, accessed on December 16, 2017.

[58] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals, Part 1: System architecture, processes,threads, memory management,and more*, 7th ed. Redmond, WA, USA: Microsoft Press, 2017.

[59] StatCounter, "Desktop Windows Version Market Share Worldwide (Jan-Dec 2020)," [Online; https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/#monthly-202001-202012-bar], 2021, accessed on January 05, 2021.

[60] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual–Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, Intel Corporation, Sep. 2016, online; https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. Accessed on July 10, 2020.

[61] D. Uroz and R. J. Rodríguez, "Evaluation of the Executional Power in Windows using Return Oriented Programming – dataset and artifact scripts," [Online; https://doi.org/10.5281/zenodo.4603061], May 2021, accessed on March 29, 2021.

APPENDIX

A. Software Downloaded from FossHub

List of TOP 10 software downloaded from the FossHub website (accessed on December 24, 2020): Audacity 2.4.2; qBittorrent 4.3.1; Classic Shell 4.3.1; MKVToolNix 51.0.0; IrfanView 4.56; HWiNFO 6.40; Shotcut 20.11.28; Avidemux 2.7.6; Calibre 5.8.0; Code Blocks (including compiler) 20.03.

B. Artifact Evaluation

The dataset and artifact scripts used for evaluation in this paper are freely available in [61] under Creative Commons Attribution 4.0 International license. The source code of `ROP3` is released under the GNU/GPLv3 license and freely available in [56]. The version used for reproducing the results provided in this paper is version 0.9 (see *Releases* in [56]). Complete and detailed instructions to exercise and reproduce our experiments are given in the dataset.