[m]allotROPism: A Metamorphic Engine for Malicious Software Variation Development⋆

Christos Lyvas[a,1], **Christoforos Ntantogian**[b,2], **Christos Xenakis**[c,1]

[1]Department of Digital Systems, University of Piraeus, Greece
[2]Department of Informatics, Ionian University, Greece

Received: date / Accepted: date

Abstract For decades, code transformations have been a vital open problem in the field of system security, especially for cases like malware mutation engines that generate semantically equivalent forms of given malicious payloads. While there are abundant works on malware and on malware phylogenies classification and detection in general, the fundamental principles about malicious transformations to evade detection have been neglected. In the present work, we introduce a mutation engine, named [m]allotROPism, to generate malicious code deviations with equivalent semantics from a static-analysis point of view. To achieve this, we reduce the problem of generating semantically equivalent solutions of given assembly code into a decision problem, and we solve it with the aid of satisfiability modulo theories. Moreover, we leverage return-oriented programming techniques to alter the traditional execution control flow from text to stack memory segment. We have implemented our proposed mutation engine and evaluated its detection evasion capabilities. Results show that so far, our approach is undetectable against popular free and commercial anti-malware products. We release the implementation of [m]allotROPism as open source. Our intention is to provide a method to generate malware families for experimental purposes and inspire further state-of-the-art research in the field of malware analysis.

1 Introduction

In the area of software exploitation, Return Oriented Programming (ROP), proposed by Shacham [1], gained increased attention during the late 2000s as an advanced stack smashing method that could bypass a variety of malicious code execution prevention mechanisms (*i.e.*, Write⊕eXecute). In principle, ROP is a rediscovery of threaded code in which programs typically consist of a chain of addresses in the stack pointing to code chunks in the attacked binary (or its loaded libraries) each of them ending with a return assembly instruction. These borrowed code chunks are called gadgets and their "return" is, in fact, a call to the next gadget in the chain. Nowadays, the majority of exploit code at some point use ROP techniques.

A discovered zero-day vulnerability, along with its related exploit code can be sold in the Dark web for a hefty price tag [2]. Another profitable activity in the Dark web is malware development. The increased number of attacks and, above all, the professionalization of the techniques used by cybercriminals', have been at the root of the exponential proliferation of malware. In 2017, up to 285,000 new malware samples per day were registered, the highest number ever observed [3]. Despite the improvement of antivirus software, sophisticated malware uses advanced evasion techniques to bypass detection points. One of the most advanced techniques for antivirus evasion is metamorphism. A metamorphic code carries a mutation engine, in order to generate new species that share the same semantics using different instructions. While there is an abundance of available researches on malware and on malware phylogenies classification and detection in general, there is no systematic research of metamorphic engines and code transformation techniques that generate semantically equivalent forms of given malicious payloads.

This paper blends techniques from two different research areas, namely malware development and software exploitation, to design a mutation engine, which we call

⋆Thanks to the title
[a]clyvas@unipi.gr
[b]dadoyan@ionio.gr
[c]xenakis@unipi.gr

[m]allotROPism[1]. More specifically, our mutation engine is capable of generating semantically equivalent variations of an input malware in an automated manner. To achieve this, we reduce the problem of generating semantically equivalent solutions of given assembly code into a decision problem, and we solve it with the aid of Satisfiability Modulo Theories (SMT). Moreover, we leverage ROP techniques, not for its initial purpose (i.e., software exploitation), but for its ability to alter the execution control flow from `Text` to `Stack Memory Segment`. To this end, [m]allotROPism automatically transforms any given assembly shellcode to its equivalent ROP representation. The proposed mutation engine fulfills all the requirements of metamorphism, namely code substitution, register swap, garbage insertion, and code flow diffusion. In this way, [m]allotROPism can be considered a principled approach to metamorphism. Finally, we have implemented our mutation engine to evaluate it against anti-malware products. Results show that so far, our approach is undetectable by both commercial and free antivirus software. To the best of our knowledge, [m]allotROPism is the first work that combines SMT solver and ROP transformations in order to fulfill all the metamorphic characteristics without imposing restrictions. [m]allotROPism currently can handle only macOS platform's shellcodes. The reason why we chose the macOS platform is because its shellcodes strike a balance between the intrinsical simplicity of Unix-based shellcodes and the much more sophisticated shellcodes of Windows systems.

To summarise, the contributions of this work are:

– We propose a metamorphic engine that generates all the semantically equivalent ROP representations as in Section 6.
– We implement and evaluate our metamorphic engine against anti-malware products
– We make our prototype and the results[2] of our empirical analysis publicly available for experimental verification.

The rest of this paper is structured as follows: Section 2 provides an overview of the related work, while Section 3 provides all the needed background information related to our approach. Section 4 analyses the motivation of this work and provides a high-level overview of [m]allotROPism functionality. Section 5 elaborates on the [m]allotROPism mutation engine and its modules. Section 6 presents the results of our experimental evaluation and performs a comparative analysis with the related work as well as discusses possible limitations of our approach. Finally, Section 7 concludes the article.

[1]. In chemistry, the ability of an element to exist in more than one physical form without change of state is called allotropism

[2]`https://gitlab.ds.unipi.gr/systems-security-laboratory/mallotropism`

2 Related Work

A significant amount of work has focus on software exploitation and malware analysis (development or detection). Here we cover only the most recent and relevant works.

Software exploitation using ROP. Schwartz *et al.* [4] developed an exploit hardening system named Q to automate the generation of Return Oriented Programming gadgets and enhance exploit payloads. In their work, they evaluated the existing code re-usage mitigation mechanisms (Write$\oplus$eXecute - W$\oplus$X, Address Space Layout Randomization - ASLR). Their methodology is divided into several steps to fulfill its purpose. In the initial step, the user provides malicious payload that she/he wants to execute. This payload is described in Q's high-level language, named QooL. In the second step, Q discovers the necessary gadgets (to perform the operations of the malicious payload of the previous step) in a given benign program with the aid of semantic analysis. At the next phase, Q performs gadget assignment, where the gadgets are chained correctly between them. Finally, Q prints out the exploit's payload bytes.

Moreover, the work by Dullien *et al.* [5] proposes an intermediate language (IL) together with a set of algorithms to automatically find ROP gadgets. Using the IL, the proposed gadget-discovery algorithms are not limited to a specific architecture. This paper is one of the very few works that consider ROP gadget discovery on ARM and RISC architectures, while the majority of the related work focuses on x86 and x64 architectures.

While the traditional use of ROP is software exploitation (*i.e.*, bypass non-executable stack and heap), there are some previous works that have proposed ROP for benign purposes. For example, Ma *et al.* [6] propose ROP for software watermarking. The proposed ROP-based watermarking is able to transform watermarking code into ROP gadgets and build them in the data region. Once triggered using a secret message, the pre-constructed ROP execution will recover the hidden watermark message. The proposed method ensures that the watermarked program does not have an explicit code stream that belongs exclusively to watermarking. Instead, the authors use operating system libraries to borrow the ROP gadgets, preventing detection by software analysis. Moreover, RopSteg [7] has been proposed for program steganography. The latter is a variation of software obfuscation but it differs from it since in program steganography the instructions are hidden instead of being transformed. RopSteg achieves the protection of selected code by generating equivalent ROP gadgets and blending them into the executable. Finally, Mu *et al.* [8] propose ROPOB, a code transformation technique to obfuscate control flows using ROP. The main contribution of ROPOB is that due to the use of ROP to complete control flow transfer, static reverse engineering methods cannot discover the real control flow, even though they can disassemble software correctly. Although the authors of all the aforementioned works have implemented their proposed methods to evaluate their effectiveness, none of them are available publicly on the internet (*i.e.*, source code or in the form of an executable).

In a more recent work Weidler *et al.* [9] leverage the ROP technique to present arbitrary code execution attacks against Internet of Things (IoT) devices. The authors in their work showed that even small sections of executable code are enough to create ROP gadgets that can erase and reprogram the flash memory of an embedded device. They concluded that the practice of loading the on-chip ROM with peripheral driver libraries and other potentially unused code makes it a prime target for ROP attacks.

Malware development or detection. Mohan *et al.* [10, 11] developed a metamorphic obfuscator called Frankenstein, which is able to reassemble a given malware with code fragments entirely from other benign programs. Authors' motivation was the creation of malware variations from benign pieces of randomly selected binaries residing in a system. Their goal is to avoid Signature Matching (syntax–based heuristics) detection. They reduce the problem of generating mutations into a searching problem. Their proposed method is able to search for segments of code found in benign binaries and compare them semantically with the given malicious instructions. Finally, Frankenstein performs suitable code arrangements to construct the final payload.

In the same direction as Frankenstein, Poulios *et al.* [12] use Return Oriented Programming (ROP) in order to merge malicious shellcodes in return-oriented form into benign programs. ROPInjector takes a shellcode and a benign binary as input which is going to be trojanized. Technically the proposed method can find semantically-equivalent ROP gadgets and compare them with the given shellcode assembly commands. If the step of the gadget evaluation fails, ROPInjector is able to inject all the needed code left in Return Oriented Programming form by extending the `Text Segment` of the benign binary. The purpose of that work was to hide malware within a benign program (trojanization) and evade antivirus detection by leveraging the transformation of shellcode into its ROP representation.

Regarding malware detection, a recent method named BinSim [13] works almost in the opposite way from our proposed [m]allotROPism. More specifically, BinSim is designed for binary diffing analysis, in order to detect similarities or differences across multiple code blocks. The main idea of BinSim is first to do a sequence call alignment to obtain a list of matched system call pairs. Next, starting from the matched system call arguments, BinSim performs a well-known technique for software debugging named dynamic backward slicing to identify instructions that affect the argument both directly and indirectly. For each produced slice, symbolic formulates, which capture the data and control flow, are extracted. Finally, BinSim uses an SMT solver to verify whether two symbolic formulas are equivalent.

Code Synthesis. Jha *et al.* [14] define the concept of *Oracle-based* software synthesis where loop-free code segments can be generated automatically with the aid of *Satisfiability Modulo Theories (SMT)* solver. The outcome of their proposed method was the development of the *oracle-guided* synthesis framework named Brahma, which leverages SMT solvers in two ways. At the initial step, Brahma starts from a random value to build the equivalent code segment with the aid of SMT solvers. In the following step, from the random value that a variable holds they feed the SMT solver with the desired program and the variable that holds the random value of initial step to define the exact value that they differ. Then from the distinguishing value, they feed the SMT solver to generate the code snippet that is capable to cause the change from the random value to the distinguishing one. The above-mentioned steps are repeated until the generated code is equivalent to the desired one. The authors in their research used Brahma to successfully deobfuscate segments of existing malware code due to the fact that from simplified code snippets that had been generated by their proposed method they could understand what the malicious code was programmed to do.

In the context of non-academic work, Rolles [15] introduced Synesthesia a framework for creating machine code programs that fulfill a specific given encoding. Synesthesia [3] leverages SMT solvers in order to generate shellcodes under encoding restrictions. Synesthesia's inputs are a specification of the desired functionality of shellcode and specifications of allowable encodings. Synesthesia can be used for the elimination of unwanted opcodes that contain *NULL bytes* that interrupt the execution of shellcodes or decrease the size of payloads in order to fit in corrupted *Stack Segment* memory. Synesthesia is a collection of Yices [16] (an SMT solver language) scripts.

In the same direction as Brahma [14], Blazytko *et al.* [17] focused their research on malware deobfuscation. They proposed a generic approach for trace simplification based on program synthesis with the aid of SMT solvers and emulation guided by Monte Carlo Tree Search (MCTS) to obtain the semantics of different kinds of obfuscated code. They implement a prototype of their proposed method named Syntia[4]. Based on I/O samples from assembly code as input, Syntia can apply MCTS-based program synthesis to compute a simplified expression that represents a deobfuscated version of the input. Syntia initially analyzes assembly code by utilizing a disassembler then it uses the SMT solver for expression simplification. Syntia expects from the user a memory dump and a start and an end address as input. Then, it emulates the program and outputs

[3]`https://github.com/RolfRolles/SynesthesiaYS`
[4]`https://github.com/RUB-SysSec/syntia`

the instruction trace. Given this trace, the user can define its own rules for trace dissection. Otherwise, Syntia dissects the trace at indirect control transfers. Additionally, the user has to decide if register and/or memory locations are used as inputs/outputs and how many I/O pairs shall be sampled. Syntia traces register and memory modifications in each trace window, derives the inputs and outputs, and generates I/O pairs by random sampling. Finally, Syntia was tested against ROP gadgets, and it was able to synthesize their semantics of ROP gadgets with very high precision.

3 Preliminaries

3.1 Return Oriented Programming (ROP)

ROP [1] is a technique that enables attackers to perform malicious computing operations without injecting arbitrary code into the address space of the target process. The main motivation behind ROP is to bypass the restriction of code execution prevention mechanism (also known as Write$\oplus$eXecute). ROP is Turing complete meaning that it can perform any possible computation in a given instruction set (x86, x64, ARM, MIPS). In software–based attacks, ROP takes place after the attacker succeeds in hijacking the execution control of a program (typically through a buffer overflow). In particular, after controlling the instruction pointer register, an attacker executes short sequences of instructions that end with the ret instruction known as ROP gadgets. These instructions are found inside the binary image or the loaded libraries. In most instances of software–based attacks, the execution of the sequence of the ROP gadgets (also known as ROP chain) aims at calling an API (in Windows–based systems) or a system call (in Linux based systems) to flag the stack or the heap segment as executable. In this way, the attacker can execute the main functionality of the malware (also known as shellcode). Selecting the ROP gadgets and building the ROP chain automatically (or even manually), in order to execute an arbitrary computation is a challenging task. To exemplify, gadgets that perform conditional/unconditional jumps may break the correct order of ROP chain execution. Gadgets that insert or remove values from the Stack Segment may disrupt the sequential execution of the ROP chain. Despite the difficulties of building a ROP chain, several research works have achieved to semi-automate the process of building a ROP chain as further discussed in Section 6.3. On the defensive side, security mechanisms such as ASLR [18], control flow integrity protection [19], etc. have significantly raised the bar for attackers, since they (*i.e.*, the attackers) must also bypass these defensive mechanisms [20].

3.2 Satisfiability Modulo Theories (SMT)

Boolean Satisfiability Problem (SAT) is a decidable problem proven to be NP-complete [21]. Technically, is the problem of determining if a set of variables for a given Boolean formula can be persistently expressed by the values True (satisfiable) or False (unsatisfiable). In other words, it is able to decide whether there exists a satisfying solution for a given well-formed formula in propositional logic. Satisfiability modulo theories (SMT) extend boolean satisfiability (SAT) by adding equality reasoning, arithmetic, bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability or the validity of formulas in these theories [22].

SMT solvers have already been used in a variety of scientific domains of informatics. Park *et al.* [23] in their research presented a formal verification tool for the Ethereum cryptocurrency Virtual Machine (EVM) bytecode with the aid of SMT solvers. Vanhoef and Piessens [24] combined an SMT solver with symbolic execution in order to efficiently simulate cryptographic primitives. Using this approach, they were able to detect several flaws in implementations of the WPA2 4-way handshake. Using SMT solvers is not a new technique in software security. In particular, as analyzed by Vanegue *et al.* [25] it has found applications in exploitation development and software security. Mostly, they have been used to discover vulnerabilities and to evaluate ROP gadgets, in order to check for an equivalent ROP gadget of given shellcode instructions. The motivation of the related researches [4, 5] is to provide full ROP compilers able to automatically identify ROP gadgets and then combine them to create a ROP chain that is equivalent to the given exploit code.

Moreover, as we already mentioned in the previous section code synthesis [26] can be achieved with the aid of SMT solvers. Brahma [14] and Syntia [17] can efficiently synthesize programs for software deobfuscation while Synesthisia [15] leverages code synthesis to escape unwanted encodings from an exploitation development standpoint. We analyze the technical insights of Brahma, Syntia and Synesthisia in detail in our comparative analysis in Section 6.3.

In our research, we leverage SMT solver to transform given assembly operations. Let's assume a subset of the x86-x64 instruction set with only `add`, `sub` and `mov` commands and `rax`, `rbx` as it's only registers. Let's assume now the input command `add rax, 0x2` where `rbx` is equal to `0x7` and `rax` to `0x5`. We express the operation in its mathematical representation as $rax = rax + 0x2$. At that point, we have programmed the SMT solver to get the mathematical representation the value of the affected register (`rax`) before (`0x5`) and after the execution(`0x7`) in order to generate all the possible operation of any operand, address, register or numerical value over the affected register (`rax`) that is going to set it equal to `0x7` after the execution. Commands such `mov rax, 0x7`, `mov rax, rbx` are equivalent to the initial one in order to set `rax` equal to `0x7` after the execution.

4 Motivation and Overview

4.1 Motivation

Modern malware has tried to use various mutation techniques to avoid detection. For instance, polymorphism is the use of encryption to hide the malicious code, while botnet infections can update their code through command and control channels. Unsurprisingly, anti-malware products have developed an array of detection technologies to identify malware despite mutations. One approach is semantic detection, which classifies programs as malicious or benign based on their behavior rather than their syntax. However, semantic detection is ineffective against previously unseen behaviors, such as zero-days. Moreover, it cannot reliably detect time bombs, which unleash their malicious behaviors only after days or weeks of waiting; and is impractical to apply indiscriminately to the millions of software programs on large networks. As a result, state-of-the-art malware detection still relies heavily upon syntax-based heuristics (*i.e.*, signature matching) as a first step toward identifying suspicious programs worthy of greater analysis. That is, in polymorphic malware the original code is revealed at runtime; hence they can be detected by analyzing the memory, while command and control updating is potentially detectable using signatures at the network level (*i.e.*, intrusion detection systems).

On the other hand, metamorphic malware does not share these limitations. Metamorphic engines can be defined as systems that are able to provide code reforms (mutations) of malicious prototype payloads with equivalent semantics [27]. The purpose of a mutation engine is the generation of malware variations that cannot be discovered by signature–based antivirus. According to O'Kane *et al.* [28] metamorphic engines usually perform one or more of the following transformations:

1. **Code Substitution.** Where opcodes substitute initial equivalent opcodes able to perform the same operations.
2. **Garbage Insertion.** Where benign instructions are inserted such as no operations in the initial malicious payload.
3. **Register Swapping.** Where registers have been swapped between malware variations.
4. **Control Flow Diffusion.** Where malware developers insert subroutines to reorder the call sequence with the aid of jumps and calls.

In theory, metamorphic malware may be undetectable. However, real-world malware is typically armed with only a limited set of mutations (due to limited code transformations). Once this set is known, it can be automatically reversed or normalized to detect all variants. Moreover, malware can partially achieve metamorphism by fulfilling one of the above four characteristics not necessarily all of them at the same time. This means that the generated variants may not have large deviations and by using statistical analysis, signatures can be extracted. To the best of knowledge, [m]allotROPism is the first work that fulfills all the above–mentioned metamorphic characteristics without imposing restrictions. In particular, [m]allotROPism employs an SMT solver to generate equivalent forms of malware based on a given assembly payload to achieve Code Substitution and Garbage Insertion. Moreover, using ROP it achieves Control Flow Diffusion and Register Swapping characteristics. Thus, [m]allotROPism can be considered a principled approach to metamorphism.

4.2 Architecture

In this section, we provide a high-level overview of [m]allotROPism. In the present research, we focus on a specific category of malicious software called shellcode[5]. According to Spafford [29], the shellcode is defined as malicious software that is capable of establishing TCP (Transmission Control Protocol) connections across an infected system and a system controlled by an attacker. This technique gives the intruder the ability to remotely execute OS (Operating System) commands on the infected system. As shown in Figure 1, [m]allotROPism takes as an input one shellcode and outputs all possible variations of ROP based shellcodes. The metamorphic engine of [m]allotROPism consists of 5 distinct modules:

– **Basic-Block Calculation.** The initial shellcode for which we want to generate the ROP equivalent variations is processed and divided into smaller code sections named basic blocks (Pseudocode 1 Line 2). As we analyze below, these basic blocks allow the creation of equivalent ROP transformations.
– **Command Emulation.** This module executes the commands inside the basic blocks from the previous Basic Block Calculation. The result of every operation after the emulation is saved in an array called Machine State (Pseudocode 1 Line 4 to Line 8). In particular, a machine state holds the register values after every command emulation.
– **SMT Solver.** Now, the output of command emulation module (*i.e.*, machine state) becomes the input of the

[5]Introduced initially to execute `Unix Shell` commands and it is usually written in machine code.

SMT solver module (Pseudocode 1 Line 12). More specifically, we feed the SMT solver with two consecutive machine states (*i.e.*, before and after the execution of an instruction), and the solver outputs equivalent instructions. To achieve this, for each assembly instruction we have developed a corresponding mapping to an SMT formula. It is important to mention that the SMT solver is not limited only to output one solution (*i.e.*, equivalent instruction) for a given instruction, but it is also capable to generate a combination of instructions as a solution (*i.e.*, it can produce the results of any transition from one machine state to another with a variety of equivalent assembly operations). As we will analyze in section 5.3, the SMT solver is the responsible module that fulfills Code Substitution and Garbage Insertion requirements of metamorphism.

– **Validation.** The SMT solver outputs equivalent commands (Pseudocode 1 Line 13 to Line 19) that satisfy the mathematical expressions, but it is unaware of the actual memory operations. This means it can generate results that are not valid. Therefore, in order to eliminate side effects produced by flag registers and memory operations, we emulate (Pseudocode 1 Line 15) every possible solution of the previous module (*i.e.*, SMT Solver), in order to discard possible instructions that violate memory-related operations.
– **ROP Transformation.** This module takes as input all the validated equivalent instructions (from the validation module) as well as the basic blocks (from the basic blocks calculation module) and outputs ROP shellcodes (Pseudocode 1 Line 23), which are all variants of the initial shellcode. More specifically, this module is able to transform all the basic blocks with all the variations produced by the SMT solver module into ROP gadgets. Next, the generated gadgets are chained and all the possible ROP representations of the input shellcode are produced. As we analyze in section 5.5, due to the ROP Transformation module, [m]allotROPism fulfills the Register Swapping and Control Flow Diffusion requirements for metamorphism.

5 Mutation Engine

In this section, we analyze in detail the mutation engine modules of [m]allotROPism. Our aim is to combine our analysis with step-by-step examples for a better understanding of the presented techniques. To this end, we will use a code snippet (see Listing 1) from a real-world x64 shellcode for macOS that performs a reverse TCP connection between an infected and an attacker-controlled system (see Appendix for the full code of the considered shellcode). As we will analyze in section 6.2, the same shellcode will be used for our experiments.

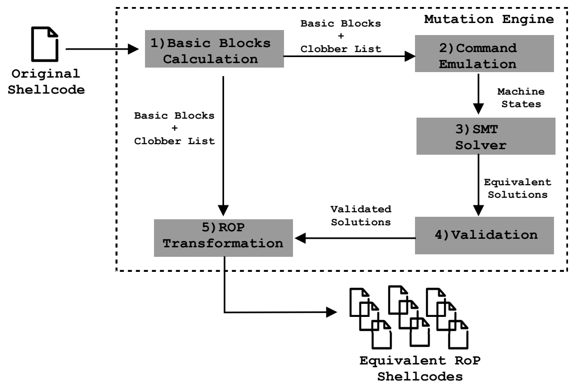

Fig. 1: [m]allotROPism High–Level Overview.

5.1 Basic-Block Calculation.

In the initial step a given shellcode is processed and divided into basic blocks. This process ensures that specific assembly operations (such as syscall, pop-push, cmp, jmps, call and memory operations over registers) unsuitable to be expressed as ROP equivalent operations will cause no disruption or inconsistencies during the ROP transformation. The rules for the creation of the basic block is as follows:

1. A separate block is required for syscall instructions.
2. A separate block is required for Push-Pop instructions.
3. A separate block is required when instructions belonging to a function are executed.
4. Branch instructions (e.g., `cmp, je, call`) are not included in blocks.
5. All other instructions are grouped into one block (maximum number of instructions per block is 6).

Moreover, this module for each basic block calculates the so-called *Clobber List*. The latter holds the registers that change their value by the instructions of the block. As we analyze below, using the *Clobber List*, we are able to identify all the free-to-use registers for each basic block, and perform the final transformation into ROP representation.

To exemplify, the code in Listing 1 is divided into basic blocks and the Clobber List is computed (see Listing 2). In particular, Block 1 is created for the first instruction `xor rsi, rsi`, while Block 2 is created for the instructions `mov rax, r8` and `mov rdi, r12`, since a new function starts from line 2 in listing 1. Next, a new block (*i.e.*, Block 3) is created for the `syscall` instruction. Block 4 includes only the `inc rsi` instruction, since it is the last command of the function. Finally, the last two instructions (*i.e.*, `sub r8, 0x1f` and `mov rax, r8`) are included

```
xor rsi, rsi
function:
    mov rax, r8
    mov rdi, r12
    syscall
    cmp rsi, 0x2
    inc rsi
    jbe function
sub r8,0x1f
mov rax, r8
```

Listing 1: Code Example.

```
Block 1:{
    xor rsi, rsi
}
Clobber List 1: {rsi}
function:
Block 2:{
    mov rax, r8
    mov rdi, r12
}
Clobber List 2: {rax, rdi}
Block 3:{
    syscall
}
Clobber List 3: {rax, r11}
cmp rsi, 0x2
Block 4:{
    inc rsi
}
Clobber List 4: {rsi}
jbe function
Block 5:{
    sub r8,0x1f
    mov rax, r8
}
Clobber List 5: {r8, rax}
```

Listing 2: Basic Blocks and their corresponding Clobber Lists.

in Block 5. Note that the commands `cmp rsi, 0x2` and `jbe function` are not included into blocks since they are branch instructions. Also, the computed Clobber Lists include the registers that change their values inside the basic blocks. For instance, in Block 2, the corresponding Clobber List includes the registers `rax` and `rdi`, since these two registers will be modified by the instructions of Block 2.

5.2 Command Emulation

The first step of the emulation module is to set all registers to zero because all the registers (except stack registers and instruction register) holds unknown to us values. We chose to initialize them to zero because they are all available for use at the beginning of every program execution. After the initialization of registers, every instruction in each block is executed one after another. Whereas the same sequential execution applies to instructions across basic blocks. The command emulation module stores the result of each execution (*i.e.*, register values) into an array called *Machine State*. Table 1 shows the results of the emulation of the basic blocks for the considered example (see Listing 2). Please note that for simplicity reasons we include in Table 1 only the registers that are modified in our example (*i.e.*, `rax, rsi, rdi, r8`). However, the command emulation module stores the values of all registers into the machine state. Also, note in the initial state 0, not all registers have value to `0x0`, because the considered example is only a code snippet of a shellcode; thus the first command (*i.e.*, `xor rsi, rsi`) is not the first one emulated, but we defined it as initial state 0 for our considered example. Apart from the machine state, the command emulation stores also the execution flow trace (see the third column of Table 1). That is, it records the sequence of the executed instructions during emulation taking into account conditional branches, loops, and function calls.

To better understand the notion of Machine States as depicted in Table 1 consider the following example. For the transition from state 1 to 2, we observe that the register `rax` changed its value from `0x2000062` to `0x200005a`. This occurred due to the fact that the emulated instruction was `mov rax, r8`. Thus, after this command, the value of `rax` should be equal to the value of `r8`, which is `0x200005a` in state 1.

5.3 SMT Solver

After the successful emulation of the shellcode, the SMT solver is responsible for calculating all the possible equivalent assembly operations that produce the desired result in the left-handed register, which is defined by its position in the Machine State. The mutation engine is not limited to one-to-one code transformations but is also capable to produce the results of any transition from one state to another with a variety of equivalent assembly operations. In order to reduce the complexity of the proposed examples, we chose to present only one to one code transformations.

As we mentioned in 4.1, [m]allotROPism fulfills the four metamorphic malware requirements. Based on the SMT solver we achieve `Code Substitution` and `Garbage Insertion` characteristics. Code Substitution is performed by the SMT solver, which will provide a set of equivalent commands (that can be used for Code Substitution). To elaborate more, we consider again the provided example. For instance, for the case of transition from state

Machine State	**Emulated Command**	**Trace execution flow of Listing 1**	**rax**	**rsi**	**rdi**	**r8**
0	-	-	0x2000062	0x0	0x3	0x200005a
1	xor rsi, rsi	Line 1	0x2000062	**0x0**	0x3	0x200005a
2	mov rax, r8	Line 2 (Loop 1)	**0x200005a**	0x0	0x3	0x200005a
3	mov rdi, r12	Line 3 (Loop 1)	0x200005a	0x0	**0x3**	0x200005a
4	syscall	Line 4 (Loop 1)	**0x0**	0x0	0x3	0x200005a
5	inc rsi	Line 5 (Loop 1)	0x0	**0x1**	0x3	0x200005a
6	mov rax, r8	Line 2 (Loop 2)	**0x200005a**	0x1	0x3	0x200005a
7	mov rdi, r12	Line 3 (Loop 2)	0x200005a	0x1	**0x3**	0x200005a
8	syscall	Line 4 (Loop 2)	**0x1**	0x1	0x3	0x200005a
9	inc rsi	Line 5 (Loop 2)	0x1	**0x2**	0x3	0x200005a
10	mov rax, r8	Line 2 (Loop 3)	**0x200005a**	0x2	0x3	0x200005a
11	mov rdi, r12	Line 3 (Loop 3)	0x200005a	0x2	**0x3**	0x200005a
12	syscall	Line 4 (Loop 3)	**0x2**	0x2	0x3	0x200005a
13	inc rsi	Line 5 (Loop 3)	0x2	**0x3**	0x3	0x200005a
14	sub r8, 0x1f	Line 6	0x2	0x3	0x3	**0x200003b**
15	mov rax, r8	Line 7	**0x200003b**	0x3	0x3	0x200003b

Table 1: Machine States of the considered code example (bold text indicates a changed value)

13 to state 14 based on table 1, the solver will try to find all the possible solutions that set register r8 from value `0x200005a` to `0x200003b`. In this case, the possible assembly operation is not only based on arithmetic calculations but also based on the values of the other registers if that is possible. Thus, possible solutions that set register `r8` equal to `0x200003b` are `sub r8, 0x1f`, `sub r8, 0x0ffffffffffffe1`, `xor r8, 0x61` and `mov r8, 0x200003b` (see Table 2).

Moreover, because of the SMT solver our method is able to insert garbage into generated mutations in terms of no-operations in explicit cases where the execution of commands does not affect the value of any register. For instance, in transition from the state 0 to state 1, which the instruction `xor rsi, rsi` is emulated (see Table 1), all values remain equal. In this case, the SMT solver will produce all the solutions that do not change the values of the registers in the Machine State for the transition from state 0 to state 1. Thus, some of the generated equivalent results (which in our context are considered garbage instructions) are `sub rsi, r13` (since both `rsi` and `r13` are equal to zero, the instruction `sub rsi, r13` will not change `rsi`) or `xor rsi, r13` (see Table 2).

As a final note, the SMT solver may not generate equivalent instructions for every machine state transition. For instance, for machine state transition from state 4 to 5 (see Table 1), the solver was not able to find an equivalent instruction for the command `inc rsi` (see Table 2). Note that the instruction `add` is different from `inc`, because the former modifies the Carry flag while the latter does not, and therefore they cannot be used interchangeably. The same is also true for very specific commands, such as `syscall` that do not have equivalent commands.

5.4 Validation

This module can be divided into phase I and phase II validation. In phase I, the module will validate the equivalent instructions generated by the SMT solver, in order to ensure that instructions that use memory values will be discarded. Moreover, in phase II the module will actually compile and run the transformed shellcodes to verify that the transformations do not cause unexpected errors caused by flag registers, memory miscalculations, etc.

More specifically, regarding phase I validation, the main goal is to ensure that that commands (generated by the SMT solver) with arithmetic operations over stack registers or values will be discarded. The SMT solver generates such instructions because it is unaware of the memory randomization, which prohibits the use of these instructions. For instance, lets assume that `rsp` holds a random memory value (*i.e.*, RandMemVal) and the next command is `mov rax, rsp`. In this case, the solver may generate the following possible equivalent instruction: `mov rax, RandMemVal`. For the solver, the above-mentioned command is indeed correct and will set the register `rax` to hold the same memory value with `rsp`. However, the value of RandMemVal is not constant; it is modified each time the shellcode is re-executed, due to address randomization. Thus, the command `mov rax, RandMemVal` will not be valid for subsequent executions of the shellcode and will be discarded.

After phase I, the phase II validation takes place to ensure that side effects caused by flag registers, miscalculations in the memory and other factors will not cause unexpected behavior. In this phase, the validation module replaces the first instruction of the initial shellcode with the corresponding equivalent assembly commands generated by the SMT solver. Every uniquely transformed shellcode is compiled and executed. For each non-successful execution of the transformed shellcode, the validation module discards the equivalent assembly command that was used in the shellcode. When all the possible solutions for the first instruction of the shellcode are tested, the validation module continues with the next instruction of the shellcode. This procedure is repeated until all possible solutions for all shellcode instructions are validated.

5.5 ROP Transformation

The last module of the metamorphic engine takes as input the verified instructions as well as the basic blocks and the corresponding Clobber Lists (generated by the basic block calculation module) and taking into account the execution trace flow, generates the ROP gadgets. As we have mentioned, the latter are instructions that end with the command `ret`. Thus, the ROP Transformation module appends the instruction `ret` at the end of every block to create the ROP gadgets.

Although the conversion of blocks to ROP gadgets is straightforward (*i.e.*, append a `ret` instruction), special care should be taken to cope with loops (e.g, FOR/WHILE statements) and conditional branches (e.g., IF statements). ROP is Turing Complete and by this we fully take advantage of its capabilities to deal with loops and conditional branches. To this end, we define a number of Control Flow Maintenance gadgets, which are special ROP gadgets that are generated by the ROP Transformation module, and that convert loops and conditional branches into ROP form. Evidently, the registers that will be used in these gadgets should be free, in order not to affect the rest of the ROP execution. We define these registers as Control Flow Maintenance registers. Since for each basic block, the used registers are collected in the Clobber list (see section 5.1), we can easily identify the free-to-use registers.

To elaborate more, in order to convert loops and conditional branches into ROP form, we need three Control Flow Maintenance registers to perform a loop and two Control Flow Maintenance registers to perform a conditional branch. In the following, we analyze with our considered example why we need three free registers for the creation of ROP based loops (the logic for dealing with conditional branches and the need of two registers is similar and not analyzed). First, let us recall the example of code Listing 1. We can identify that in line 2 of Listing 1 there is a loop until line 8 where it ends. In the between there is the inner block (lines 3-7) of the loop and a condition check (in line 6) based on the left side register `rsi`. From a basic blocks point of view (see Listing 2), the step size is the Block 1, the inner loop's operations are the Blocks 2 and 3 and step change operation is the Block 4.

Now we will convert the above-mentioned loop to its ROP counterpart (see Listing 3). In this example, we do not consider any code substitutions. The first gadget (*i.e.*, Gadget 1) is created simply by appending in Block 1 the `ret` instruction. The next three gadgets (*i.e.*, Gadget 2, 3 and 4) are Control Flow Maintenance gadgets and we need three free registers to create them. Specifically, we need: i) one register to hold how many gadgets the loop is going to slide back (in our example 80 bytes). In the considered example we select `rbx` and we initialize its value in the Control Flow Maintenance Gadget 2; ii) one register that holds the upper repetition bound. In our example we select `r15` to be used and we initialize it in Control Flow Maintenance Gadget 3; iii) One register that is responsible for the branch execution in the loop's inner body. We select `r14` and we initialize it in Control Flow Maintenance Gadget 4.

After the sequential execution of the above–mentioned gadgets the loop inner body is going to be executed for the first time (*i.e.*, Gadget 5 to Gadget 8, which are directly created by Blocks 5 to 8 by appending the `ret` instruction). The following gadgets are used to maintain the loop and they are generated using the three same registers that we initialized previously (*i.e.*, registers `rbx, r15` and `r14`). In particular, Gadget 9 is responsible to check if the step size (`rsi`) has reached its upper bound by performing the operation `xor r15, rsi`. Then the complement of the upper bound register (`r15`) is calculated by executing the operation `neg r15`. Gadget 10 performs the addition of `r14` register and the carry flag which was set in the previous gadget, because of the negation of a positive number. Gadget 11 is responsible to calculate a negative number; thus, the next gadget (Gadget 12) is not going to alter the value of the register that holds the stack slides. Finally, Gadget 13 will negate the register that holds the loop's upper bound and Gadget 14 will move the `rsp` registers 10 positions, which means that the ROP execution will return in the Gadget 4. In this way, we achieve to perform a ROP based loop. The previous operation will be repeated until the `rsi` register is going to be set in its upper bound limit. Note that Gadget 9 will set the result of zero in the register that holds the loop's upper bound and the negation over zero will not cause any alterations of the register in the Gadget 10 nor the negation over zero in the Gadget 11. Finally, Gadget 12 will set the value zero in the register that holds the stack slides and Gadget 14 will perform subtraction over zero and the execution will continue to Gadget 15, which is the ROP counterpart of Block 5.

Finally, the ROP Transformation module pushes the addresses of the created gadgets into the stack to execute the created ROP chain using the `push` or `mov` instructions. This is a typical procedure for every ROP execution code [1].

```
Gadget 1:{xor rsi, rsi; ret;}
Gadget 2:{mov rbx, 80; ret;} ; Control Flow Maintenance
     Gadget
Gadget 3:{mov r15, 0x3; ret;} ; Control Flow Maintenance
      Gadget
Gadget 4:{xor r14, r14; ret;} ; Control Flow Maintenance
      Gadget
Gadget 5:{mov rax, r8; ret;}
Gadget 6:{mov rdi, r12; ret;}
Gadget 7:{syscall; ret;}
Gadget 8:{inc rsi; ret;}
Gadget 9:{xor r15, rsi; neg r15; ret} ; Control Flow
     Maintenance Gadget
Gadget 10:{adc r14, r14; ret;} ; Control Flow
     Maintenance Gadget
Gadget 11:{neg r14; ret;} ; Control Flow Maintenance
     Gadget
Gadget 12:{and rbx, r14; ret} ; Control Flow Maintenance
      Gadget
Gadget 13:{neg r15; ret;} ; Control Flow Maintenance
     Gadget
Gadget 14:{sub rsp, rbx; ret;} ; Control Flow
     Maintenance Gadget
Gadget 15:{sub r8, 0x1f; mov rax, r8;}
```

Listing 3: One Possible ROP Transformation Example.

Last but not least, due to this module (*i.e.*, ROP Transformation) [m]allotROPism fulfills the other two metamorphic malware characteristics. That is, we achieve Register Swapping and Control Flow Diffusion properties (recall that using the SMT solver we have Code Substitution and Garbage Insertion). More specifically, by selecting all the possible permutations of free registers (for the generation of gadgets to maintain loops and conditional branches as it was described above), we fulfill the Register Swapping requirement. For instance, in the considered example we selected the set (`rbx,r15,r14`). These registers can be swapped with another set such as (`r14,rcx,r15`). Moreover, by generating ROP shellcodes from all the possible permutations of the generated gadgets we achieve the property of Control Flow Diffusion. Note that this is possible because the execution flow of ROP gadgets (also known as ROP chain) is not sequential. As a matter of fact, it is well known that the alignment of the ROP gadgets in the text section is irrelevant to the actual execution flow. This happens because in ROP, the role of the `rip` register is replaced by the `rsp` register of the stack. This means that the actual execution flow is based on how the memory addresses of the gadgets are placed in the stack.

Algorithm 1 Mutation Engine

```
 1: procedure MUTATIONENGINE($program$)
 2:     $basicBlocks := basicBlock(program)$
 3:     $clobberList := usedRegisters(basicBlocks)$
 4:     $state_0 := [0, 0, \dots, Rand, Rand]$
 5:     for all $command \in program$ do
 6:         $expretion := decode(command)$
 7:         $state_i := emulate(expretion, state_{i-1})$
 8:         $machineState_i := machineState \cup (state_i, state_{i-1})$
 9:     end for
10:     $state'_0 := [0, 0, \dots, Rand', Rand']$
11:     for all $statePair \in machineState$ do
12:         $solutions := SMT(statePair)$
13:         $substitutions := []$
14:         for all $solution \in solutions$ do
15:             $state'_i := emulate(solution, state'_{i-1})$
16:             if $(state_i = state'_i) \cap (solution \neq expretion)$ then
17:                 $substitutions := substitutions \cup encode(solution)$
18:             end if
19:         end for
20:         $transformations := transformations \cup substitution_i$
21:     end for
22:     for all $transformation \in transformations$ do
23:         $mutations := mutations \cup ROP(transformation, basicBlocks, clobberList)$
24:     end for
25: end procedure
```

6 Evaluation

6.1 Generated Mutations

First, we derive equations that allow us to calculate the total number of the generated equivalent ROP shellcodes. We argue that all equivalent ROP shellcodes are equal to the total Code Permutations multiplied by the total number of Control Flow Diffusions multiplied by the total Register Swaps:

$$Total\,ROP\,Shellcodes = |Code\,Subtitutions| \times |Control\,Flow\,Diffusions| \times |Register\,Swaps| \tag{1}$$

The total number of Code Substitutions can be defined as all the possible substitutions of all the equivalent commands for each shellcode instruction:

$$|Code\,Subtitutions| = \prod_{i=1}^{Shellcode\,lines} |Solutions_i| \tag{2}$$

where $Solutions_i$ are all the verified equivalent instructions for the specific command in line i of shellcode. The total code control flow diffusions is all the possible permutations of the produced ROP gadgets that belong in a ROP transformed shellcode:

$$|Control\,Flow\,Diffusions| = |Gadgets|! \tag{3}$$

The total register swaps for one branch is all the possible permutations of 3 registers (if the branch is a loop) or all the

Machine State	Initial Command	Code Substitutions		
		Solution 1	Solution 2	Solution 3
`1`	`xor rsi, rsi`	`sub rsi, r13`	`xor rsi, r13`	`and rsi, 0x0`
`2,6,10`	`mov rax, r8`	-	-	-
`3,7,11`	`mov rdi, r12`	`or rdi, rsi`	`or rdi, r12`	`or rdi, 0x1`
`4,8,12`	`syscall`	-	-	-
`5,9,13`	`inc rsi`	-	-	-
`14`	`sub r8, 0x1f`	`add r8, 0xffffffffffffffe1`	`xor r8, 0x61`	`mov r8, 0x200003b`
`15`	`mov rax, r8`	`or rax, r8`	-	-

Table 2: Examples of equivalent instructions generated by the SMT solver.

possible permutations of 2 registers (if the branch is a conditional branch) from the set of the free to use registers (*i.e.*, the ones that do not belong in the Clobber List. Assuming that n is the number of free registers to be used in a branch, the total number of register swaps S per branch is equal to:

$$S = \frac{n!}{(n-b)!} \quad \text{where} \quad b = \{2,3\} \tag{4}$$

Thus, the total number of register swaps is equal to S multiplied by the number of total branches in the shellcode, that is:

$$|RegisterSwaps| = S \times |branches| \tag{5}$$

6.2 Experimental Results

To evaluate the effectiveness of our proposed method we have developed a prototype of the metamorphic engine using the Python programming language. Moreover, we have utilized the Z3 solver [22] to express the SMT formulas and derive their solutions. Our implementation is open-source and free to download[6]. In our experiments, we used as input a real–world shellcode (see Appendix), which part of it was used throughout the paper to present [m]allotROPism modules in section 5. Recall that the shellcode targets macOS x64 OS. It performs a reverse TCP connection giving an attacker the ability to execute UNIX commands remotely on an affected system. The chosen shellcode of the study contains 39 lines of code. The reason why we chose for the evaluation a macOS shellcode is that it strikes a balance between the intrinsical simplicity of Unix-based shellcodes and the more sophisticated shellcodes of Windows systems. To the best of our knowledge this is the first work that considers a shellcode for the macOS system.

Our implementation was tested on an Intel Core i5 1.3 GHz CPU with 8 GB RAM. We used the nasm compiler to produce the executables from the generated shellcodes in order to execute them. We have measured the required time per generated output for a variety of given instructions. More specifically, we have selected as input assembly code snippets of various length (*i.e.*, 1, 2 and 6 assembly instructions) and operations (*i.e.*, numerical, `syscalls`, and `push`/`pop` instructions). Based on this code snippets, we measured the time required for: i) emulation; ii) to generate equivalent code substitutions of 1, 2, 3 and 4 commands (per assembly instructions of the code snippet) using the SMT solver module; iii) to generate equivalent ROP gadgets. In other words, we measured the time to provide output the emulation of the code snippets by the Validation module, the SMT solver module and the ROP Transformation module of [m]alltopROPism. The results of this performance analysis are depicted in Table 3. One can deduce that the higher number (from 1 to 4) of the computation of the equivalent commands requires more time to compete. This can be justified due to the fact that the SMT solver attempts to generate all the possible equivalent commands for an instruction based on the available registers and their values from previous states (if there are any). Additionally, the time needed for the generation of gadgets depends on the number of the given instructions (in our experiment the Code Snippet Length).

Now we describe the approach that we followed to evaluate the metamorphic properties of [m]allotrROPism and their effects on antivirus evasion. The first metamorphic property is Code Substitution. Recall that code substitutions is achieved by replacing the initial shellcode with the solutions of the SMT solver module. In total, we considered 4 different shellcodes with different level of code substitutions. That is the initial shellcode without any code modification, as well as 3 variants with different levels of code substitutions. That is, for the variant 1 we considered code substitutions for 6 out 39 lines of the initial shellcode, while in variant 2 we considered 17 out of 39 lines code substitutions. Variant 3 includes the maximum number of code substitutions that [m]allotROPism was able to achieve, which was 22 out of 39 lines of the initial shellcode. The second

[6]`https://gitlab.ds.unipi.gr/systems-security-laboratory/mallotropism`

metamorphic requirement, which is Garbage Insertion, was not considered in the experiments.

Next, we performed ROP transformations for the shellcodes of our experiment. The third metamorphic requirement that we want to evaluate its effects on antivirus evasion is Register Swapping. More specifically, the considered shellcode includes a loop, due to the jump instruction in line 33 (see Appendix). For this reason, [m]allotROPism used Control Flow Maintenance gadgets to handle this loop. As we mentioned in section 5.5, we need three registers to convert loops to ROP based loops (we defined these registers as Control Flow Maintenance registers). For the experiments, we considered two different sets. In the first one, we generated a shellcode that used the set (`r14,r11,r15`) to create the ROP based loop and next, **we swapped these registers** with the second set which was (`rbx,r15,r14`). In this way, we achieve Register Swapping. Finally, the generated gadgets can be permuted for the fourth and final metamorphic requirement, which is Control Flow Diffusion. In our experiments, we used three different levels of diffusion: 0% means no diffusion, 50% diffusion means that the half of the generated gadgets were permuted, and 100% diffusion means that all gadgets were permuted.

Moreover, we used VirusTotal to calculate the evasion rate of the generated shellcodes during the different stages of their transformations. At the time of writing the paper, VirusTotal included 59 anti-malware products. Table 4 shows the evasion results. From Table 4, we can observe that the initial shellcode was detected by 51.7% of the antivirus products. By converting the initial shellcode to its ROP equivalent with the first register set without any gadget permutation (i.e., 0%) the detection rate dropped to 0%. This is an alarming result regarding the detection capabilities of anti-malware products. By using the same register set with gadget permutation equal to 50% and 100% the detection rate remained 0%. By using the second register set for swapping and for various gadget permutation levels, the detection rate was always 0%. For variants 1,2 and 3 which we use code substitution (11, 17 and 22 out of 39 lines respectively), without ROP transformation the detection rate was stable to 47.36%. On the other hand, using ROP transformations the detection rate for the three variants dropped to 0% with or without using gadget permutation or register swapping.

6.3 Comparative Analysis

In this section, we provide a comparative analysis between our implementation and related researches (see Table 5). The work in [5] presents algorithms using an IL for gadget discovery but not for ROP compilation, in contrast to [m]allotROPism which incorporates automatic ROP transformation. On the other hand, Q [4] is a ROP compiler capable of using a small gadget search space. That is, it searches only in the vulnerable binary, but not in libraries due to randomized addresses which prohibit finding gadget. Although Q is able to discover complex ROP chains using a limited number of gadgets, it requires the use of a new language named QooL. We were not able to find documentation and examples of QooL, thus we argue that Q's has limited applicability. Finally, it seems that Q is unable in practice to generate functional payloads as reported by several pieces of research [30, 31].

The work closest to ours ROP transformation is Frankenstein [10, 11]. It is based on chaining pieces of code harvested from benign system binaries (e.g., libraries). Although the generated mutants fulfill the metamorphic properties, the authors' definition of a gadget is a more relaxed version of that used in ROP. They consider as gadgets any sequence of bytes that are interpretable as valid x86 instructions. As such it cannot be considered a pure ROP solution. The authors also do not present antivirus evasion results (possibly due to the lack of an implementation that would allow the automatic generation of mutants).

ROPInjector [12] acts as a trojan by injecting code segments capable of executing a ROP chain inside the infected host binary (*i.e.*, the ROP gadgets belong to host binary). It is the only related work that has a fully functional implementation publicly available in the Internet[7], but it relies upon several assumptions in the ROP payload generation phase. Specifically, in order to execute sophisticated operations like conditional and loop structures, it generates assembly code segments, in order to transfer the execution from Text to Stack Segment and vice versa, instead of combining plain ROP gadgets into chains. Thus, ROPInjector implements a relaxed definition of ROP transformation and not the one defined by Shacham [1]. Moreover, it generates ROP gadgets when it is unable to find them in the actual binary that wants to infect (e.g. register comparisons without being sequential followed by a conditional jump). The above characteristics of ROPInjector not only allow the creation of signatures for static detection but also clearly indicate that ROPinjector is a partial ROP compiler because it is unable to generate full ROP payloads. Compared to ROPInjector and Frankenstein, our solution is based on the pure definition of ROP as analyzed by [1] and we take full advantage of ROP's Turing completeness.

Code Synthesis. Jha *et al.* [14] proposed a deobfuscation method to simplify malicious samples with the aid of SMT solver. They developed a tool named Brahma that can be used to discover unintuitive code and for program understanding. Brahma iteratively synthesizes new programs that work correctly. It starts with a set containing just one arbitrarily chosen input. In each iteration, the procedure synthesizes a program that works correctly on the current finite

[7]`https://github.com/gpoulios/ROPInjector`

Code Snippet Length	Emulation	Number of Generated Commands				Number of Generated Gadgets		
		1	2	3	4	1	2	3
1 (numerical operations)	100 ms	502 ms	910 ms	1230 ms	1506 ms	1053 ms	-	-
2 (2 numerical operations)	140 ms	1113 ms	1910 ms	2405	3290 ms	1200 ms	1257 ms	-
2 (1 numerical operations and a `syscall`)	150 ms	505 ms	930 ms	1300 ms	1670 ms	1143 ms	1290 ms	-
2 (1 numerical operations and a `push`/`pop`)	350 ms	536 ms	923 ms	1254 ms	1705 ms	1013 ms	1264 ms	-
6 (6 numerical operations)	620 ms	3205 ms	5699 ms	7340 ms	10018 ms	3672 ms	3847 ms	4628 ms
6 (5 numerical operations and a `push`/`pop`)	610 ms	2839 ms	4810 ms	6250 ms	8179 ms	3879 ms	3595 ms	4568 ms
6 (5 numerical operations and `syscall`)	612 ms	2803 ms	4772 ms	6429 ms	8310 ms	3652 ms	3847 ms	4862 ms

Table 3: Performance of [m]allotROPism for a Variety of Code Snippet Lengths

Variation	Code Substitution (Due to SMT Solver)		Register Swapping and Control Flow Diffusion (Due to ROP Transformation)		
	Code Substitution	Detection rate (%)	Control flow maintenance registers	Gadget permutations (%)	Detection Rate (%)
Original Shellcode	0/39	51.7%	`r14, r11, r15`	0%	0%
				50%	0%
				100%	0%
			`rbx, r15, r14`	0%	0%
				50%	0%
				100%	0%
Variant 1	11/39	47.36%	`r14, r11, r15`	0%	0%
				50%	0%
				100%	0%
			`rbx, r15, r14`	0%	0%
				50%	0%
				100%	0%
Variant 2	17/39	47.36%	`r14, r11, r15`	0%	0%
				50%	0%
				100%	0%
			`rbx, r15, r14`	0%	0%
				50%	0%
				100%	0%
Variant 3	22/39	47.36%	`r14, r11, r15`	0%	0%
				50%	0%
				100%	0%
			`rbx, r15, r14`	0%	0%
				50%	0%
				100%	0%

Table 4: Generated Malicious Phylogeny (the detection rates are links pointing to VirusTotal results)

set of inputs. If such a program is found, then the procedure attempts to find a distinguishing input. If a distinguishing input is found, then it is added into the set of inputs for subsequent iterations. In all other cases, the procedure terminates. It either returns the correct program, or it notes that the components provided are insufficient for synthesizing the correct program. From a technical standpoint Brahma starts with a random input value and calculates the output of the obfuscated program on that input. Given the input/output pair of previous steps with the aid of SMT solver, they generate a candidate program. Then, with the aid of SMT solver, it checks whether a semantically different program can be generated. In this case, Brahma generates another alternative program and another distinguishing input. Then calculates the output of the obfuscated program on that distinguishing input and repeats the above steps until a semantically equivalent program to the obfuscated one is generated. Brahma deobfuscation capabilities were tested against highly obfuscated malware such as Conficker and MyDoom with satisfactory results. [m]allotROPism in contrast with Brahma uses the SMT Solver to generate code of all possible equivalent substitutions as one of the requirements of metamorphism defined by O'Kane *et al.* [28] based on input-output pairs provided by the emulator and not for code simplification. Moreover, [m]allotROPism use random values for emulation in order to discard unwanted solutions (commands) provided by the SMT Solver and not as part of the equivalent code generation procedure.

Synesthesia [15] is a framework able to generate shellcodes under specific encoding restrictions. Implementation of Synesthesia is a stand-alone compiler with three modes. The shellcode generation module under input restrictions based on shellcode's behavior. The re-compilation module of existing shellcode under input restrictions. The encode and generate decoder loops for existing, non-encoded shellcode binaries. According to its author, Synesthesia currently is an experimental release that does not fulfill the goal of being a compiler and requires intervention from the user in order to manually interpret the results.

In the same direction as Jha *et al.* [14] Blazytko *et al.* [17] proposed in their research a method that leverage emulators, SMT solvers and Monte Carlo Tree Search (MCTS) to simplify given obfuscated codes in combination with user-provided memory dumps and memory addresses. Undoubtedly, we share some implementation similarities with the publicly available prototype of Syntia [17] but for different purposes. Our method is fully automated and doesn't require execution insights. It leverages emulation to feed the SMT Solver in order to provide accurate semantically equivalent mutations of a given malware without any other user interaction in order to fulfill code substitution precondition as defined by O'Kane *et al.* [28]. As our experimental evaluation shows our mutation engine gets as input a detectable malware and provides as an outcome a very large amount of untraceable mutated malware. Syntia leverages Monte Carlo Tree Search (MCTS) based program synthesis to compute a simplified expression that represents a deobfuscated version of the obfuscated input. Syntia can operate with high precision and is also able to reconstruct the semantics from not linear ROP gadget invocations such as our method outputs but it is not clear if it can successfully reconstruct the input malware precisely in order to be detected by existing antivirus signatures. Finally, in order to be able someone to use Syntia as an automated detection mechanism against [m]allotrROPism it has to calculate all the possible variations of our mutation engine until she/he generates the input malware something that increases the overhead of the method.

6.4 Discussion

Possible detection techniques and alternative solutions. The generated variants of [m]allotROPism are subject to detection using dynamic analysis. Evidently, the ROP transformed shellcodes will execute the same system calls with the same order, therefore the execution flow is not modified. Malware detection solutions such as [13] which rely on dynamic analysis, will identify that the produced variants belong to the same malware family. However, there are several anti-dynamic solutions that can work in tandem with [m]allotROPism. For instance, one common anti-dynamic analysis solution is the use of stalling code in case the malware detects itself running in a sandbox rather than a real physical machine. That is, a malware sample can wait for a long time (e.g., by performing arithmetic calculations) before performing any malicious activity or not performing any activity at all.

There are even more robust and sophisticated solutions for disrupting dynamic analysis. Malwash [32] relies on the observation that although a malware cannot modify its system calls and the order of their execution in a binary, it can hide them within the stream of system calls that are performed on the entire operating system. Thus, rather than executing the program in a single process, Malwash automatically distributes the program across a set of pre-existing, benign processes. Another direction proposed in [33] relies on replacing a system call dependence graph to its semantically equivalent variants so that the similar malware samples within one family turn out to be different. All these solutions can work complementary to the evasion capabilities of [m]allotROPism.

Limitations. Our method inherits all the benefits and drawbacks of ROP, since it fully relies on it. More specifically, even if ROP is Turing complete, converting code structures such as loops and conditional branches to ROP can be a challenging task, since we need free registers to perform these conversions. Moreover, as shown in the experimental results, [m]allotROPism cannot replace all code lines of the considered shellcode with equivalent ROP gadgets (i.e., it converted 22 out of 39 lines). Finally, we mention that our implementation is a prototype and it is not currently functional on any other operating system than macOS.

In the current research we combine two methods (SMT Solvers and ROP gadgets) in order to fulfill formally the properties of a mutation engine [28]. Specifically, [m]allotROPism leverages the capabilities of SMT solvers in order to generate from the assembly code of a given malware as many as possible variations (transformations in terms of commands) of the input. Because of SMT our

Research	Characteristics	Limitations	Availability
Schwartz *et al.* [4]	ROP compilation for small codebase size	Requires knowledge of a new language (QooL)	No
Poulios *et al.* [12]	Binary host infection	- Partial ROP - Allows signature detection	Yes
Mohan *et al.* [11]	Discovery of equivalent instructions in system libraries	No acutal use of ROP	No
Dullien *et al.* [5]	Architecture independent gadget discovery	ROP compilation is not automatic	No
[m]allotROPism	Generation of ROP equivalent shellcodes	ROP transformation may not be always possible due to limited free registers	Yes

Table 5: Method Comparison

method it is able to insert garbage into generated mutations in terms of no-operations in explicit cases where the execution of commands does not affect their values. ROP transformation of any mutated variation of the malicious executable into purely ROP form gave us the ability to hide totally their structure while at the same time by leveraging the Turing Completeness of ROP method we were able to perform a very large amount of structure permutations in comparison with the input malware. Let us recall here that we do not find gadgets inside the memory spaces of running processes nor into libraries. Contrary to previous works, that leverage ROP gadgets found in benign programs and trigger the ROP chain typically through a buffer overflow, we have the freedom to generate whatever equivalent ROP gadget is suitable to replace an instruction. Thus, we transform an executable to any ROP counterpart. We separate a given malware into blocks according to specification rules and we convert those blocks (basic block) into ROP gadgets and we place them into mutated executable code. The pre-calculation of the size of each ROP Gadget command and their order into the executable code segment gives us the ability to know which exact address they will hold during execution. We achieve code permutations by shuffling the order of the gadgets in every generated mutation. From an ethical standpoint our intention in the current work is the generation of mutation engine able to generate malware phylogenies based on formal approaches that respect the mutation preconditions for research purpose not the generation of sophisticated malware nor the weaponization of software development. As future work, our intention is to calculate which and how mutation preconditions impact static malware detection. Our empirical evaluation shows that we generated automatically a vast amount of undetectable mutations from detectable malware in terms of malware-based signature detection. Despite the vast number of scientific researches in the field of malware detection, static and behavioral detection of malware is undecidable problems [34]. The basic principles of our SMT and ROP transformation mutation engine can be reversed and used by anti–malware engines to generate family detection signatures for malware families rather than just signatures that can only detect a specific malware sample. In other words, detection engines will be able to identify with static analysis the metamorphic properties of a malware (*e.g.*, code substitution or register swapping). It should be noted that such a detection engine, which is in essence a static analysis solution, should work in tandem with dynamic analysis complementing their detection capabilities. In general, this detection approach can raise the bar for the malicious code developers and eliminate their ability to transform an identified malware to an unknown one with basic code transformations.

7 Conclusions

In this work, we provide and analyze a formal way of generating malware families based on metamorphic preconditions with the aid of SMT solvers and ROP transformations. We design and analyze a metamorphic engine named [m]allotROPism that generates semantically equivalent variations of an input malware in an automated manner. To achieve this, we leverage SMT solvers to calculate equivalent operations for a given instruction and then algorithmically convert them into ROP representation. Since ROP is Turing-complete, we argue that we are able to execute any arbitrary computation. Contrary to previous works that leverage ROP gadgets found in benign programs and trigger the ROP chain typically through a buffer overflow, we have the freedom to generate whatever equivalent ROP gadget is suitable to replace an instruction. Thus, we are able to transform an executable to any ROP counterpart. We evaluated the proposed metamorphic engine against a set of antivirus solutions. The results showed that our proposed mutation engine is able to produce undetectable ROP variations from a real-world x64 shellcode. We believe that the antivirus industry should be aware of [m]allotROPism techniques and counteract accordingly to this potential threat. We hope also that this work will be beneficial for the research community, as we release the implementation as open–source along with our dataset (*i.e.*, the generated ROP transformations of the shellcode), so that future studies on malware detection consider also ROP as an antivirus evasion method.

Acknowledgment

This work was supported in part by:

- CUREX project of Horizon H2020 Framework Programme of the European Union under grant agreement No 826404.
- SECONDO project of H2020 MSCA RISE 2018 under grant agreement number 823997.
- SPIDER project of Horizon H2020 Framework Programme of the European Union under grant agreement No 833685

Content reflects only the authors' view and EU is not responsible for any use of the information it contains. **Conflict of Interest:** The authors declare that they have no competing interests. **Ethical approval:** This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. 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, ACM, 2007, pp. 552–561.
2. J. M. Bauer, M. J. Van Eeten, T. Chattopadhyay, Y. Wu, Itu study on the financial aspects of network security: Malware and spam, ICT Applications and Cybersecurity Division, International Telecommunication Union, Final Report, July, 2008.
3. PandaLabs, 2017 in figures: The exponential growth of malware (Accessed August 2, 2018).
   URL `https://www.pandasecurity.com/mediacenter/malware/2017-figures/`
4. E. J. Schwartz, T. Avgerinos, D. Brumley, Q: Exploit hardening made easy., in: USENIX Security Symposium, 2011, pp. 25–41.
5. T. Dullien, T. Kornau, R.-P. Weinmann, A framework for automated architecture-independent gadget search, in: 4th USENIX Workshop on Offensive Technologies (WOOT 10), 2010.
6. H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, D. Gao, Software watermarking using return-oriented programming, in: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, ACM, New York, NY, USA, 2015, pp. 369–380.
7. K. Lu, S. Xiong, D. Gao, Ropsteg: Program steganography with return oriented programming, in: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY '14, ACM, New York, NY, USA, 2014, pp. 265–272.
8. D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, L. Shi, Ropob: Obfuscating binary code via return oriented programming, in: Security and Privacy in Communication Networks, Springer International Publishing, 2018.
9. N. R. Weidler, D. Brown, S. A. Mitchell, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, R. Gerdes, Return-oriented programming on a resource constrained device, Sustainable Computing: Informatics and Systems 22 (2019) 244–256.
10. V. Mohan, K. W. Hamlen, Frankenstein: A tale of horror and logic programming, Book Reviews 2017 (02) (2017).
11. V. Mohan, K. W. Hamlen, Frankenstein: Stitching malware from benign binaries., in: 21s USENIX Workshop on Offensive Technologies (WOOT 12), Austin, TX, 2012, pp. 77–84.
12. G. Poulios, C. Ntantogian, C. Xenakis, Ropinjector: Using return oriented programming for polymorphism and antivirus evasion, Blackhat USA, 2015.
13. J. Ming, D. Xu, Y. Jiang, D. Wu, Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking, in: 26th USENIX Security Symposium, 2017.
14. S. Jha, S. Gulwani, S. A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, 2010, pp. 215–224.
15. R. Rolles, Synesthesia: A modern approach to shellcode generation (2016).
    URL `http://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty`
16. B. Dutertre, L. De Moura, The yices smt solver, Tool paper at SRI International 2 (2) (2006) 1–5.
17. T. Blazytko, M. Contag, C. Aschermann, T. Holz, Syntia: Synthesizing the semantics of obfuscated code, in: USENIX Security Symposium., 2017.
18. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, in: Proceedings of the 11th ACM conference on Computer and communications security, ACM, 2004, pp. 298–307.
19. M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, Control-flow integrity principles, implementations, and applications, ACM Trans. Inf. Syst. Secur. 13 (1) (2009) 4:1–4:40.

20. N. Carlini, D. Wagner, {ROP} is still dangerous: Breaking modern defenses, in: 23rd USENIX Security Symposium, 2014, pp. 385–399.
21. T. J. Schaefer, The complexity of satisfiability problems, in: Proceedings of the tenth annual ACM symposium on Theory of computing, ACM, 1978, pp. 216–226.
22. L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
23. D. Park, Y. Zhang, M. Saxena, P. Daian, G. Roşu, A formal verification tool for ethereum vm bytecode, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018, pp. 912–915.
24. M. Vanhoef, F. Piessens, Symbolic execution of security protocol implementations: handling cryptographic primitives, in: 12th USENIX Workshop on Offensive Technologies (WOOT 18), 2018.
25. J. Vanegue, S. Heelan, R. Rolles, SMT solvers in software security, in: 6th USENIX Workshop on Offensive Technologies (WOOT 12), 2012.
26. J. Bornholt, Program synthesis, explained (Accessed February 2, 2018).
URL `https://homes.cs.washington.edu/bornholt/post/synthesis-for-architects.html`
27. P. Szor, The art of computer virus research and defense, Pearson Education, 2005.
28. P. O'Kane, S. Sezer, K. McLaughlin, Obfuscation: The hidden malware, IEEE Security & Privacy 9 (5) (2011) 41–47.
29. E. H. Spafford, The internet worm program: An analysis, ACM SIGCOMM Computer Communication Review 19 (1) (1989) 17–57.
30. R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code, in: Proceedings of the 2012 ACM conference on Computer and communications security, ACM, 2012, pp. 157–168.
31. V. Pappas, M. Polychronakis, A. D. Keromytis, Smashing the gadgets: Hindering return-oriented programming using in-place code randomization, in: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, 2012, pp. 601–615.
32. K. K. Ispoglou, M. Payer, malwash: Washing malware to evade dynamic analysis, in: 10th USENIX Workshop on Offensive Technologies (WOOT 16), 2016.
33. J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, B. Mao, Replacement attacks: Automatically impeding behavior-based malware specifications, in: Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, Springer, 2015, pp. 497–517.
34. F. Cohen, Computer viruses: theory and experiments, Elsevier, Computers & Security 6 (1) (1987) 22–35.

Appendix

```
global start
start:
    mov r8b, 0x02
    shl r8, 0x18
    or r8, 0x61
    mov rax, r8
    xor rdx, rdx
    mov rsi, rdx
    inc rsi
    mov rdi, rsi
    inc rdi
    syscall
    mov r12, rax
    mov r13, 0x0100007f5c110101
    mov r9b, 0xff
    sub r13, r9
    push r13
    mov r13, rsp
    inc r8
    mov rax, r8
    mov rdi, r12
    mov rsi, r13
    add rdx, 0x10
    syscall
    sub r8, 0x8
    xor rsi, rsi
dup:
    mov rax, r8
    mov rdi, r12
    syscall
    cmp rsi, 0x2
    inc rsi
    jbe function
    sub r8, 0x1f
    mov rax, r8
    xor rdx, rdx
    mov r13, 0x68732f6e69622fff
    shr r13, 0x8
    push r13
    mov rdi, rsp
    xor rsi, rsi
    syscall
```

Listing 4: Shellcode.

x86-64 Instruction Set. Following the technological shift from 32-bit to 64-bit CPUs, in this paper we will consider only the modern x86-64 architecture. In x86-64 assembly there are sixteen registers: `rsp`, `rbp`, `rax`, `rbx`, `rcx`, `rdx`, `rdi`, `rsi` and `r8`-`r15`. Registers `rbx`, `rbp`, `r12`, `r15` are callee-saved registers, meaning that they are saved across function calls. In contrast `rax`, `rcx`,`rdx`, `rdi`, `rsi`, `rsp`, `r8`, `r11` are considered caller-saved registers, meaning that they may not be saved across function calls. By convention, register `rax` is used to store functions return values (if they exist and are no more than 64 bits long). For a typical function invocation, the program should place the first six integer or pointer parameters in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` to the called functions. For more than six parameters their values must be pushed onto the stack with the first argument topmost. `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.