Transforming Malicious Code to ROP Gadgets for Antivirus Evasion

Christoforos Ntantogian [1], Georgios Poulios [1], Georgios Karopoulos [2], Christos Xenakis [1*]

[1] Department of Digital Systems, University of Piraeus, Greece
[2] Department of Informatics and Telecommunications, University of Athens, Greece
[*]xenakis@unipi.gr

Abstract: In this paper, we advance research in offensive technology by proposing Return Oriented Programming (ROP) as a means to achieve code obfuscation. The key inspiration is that ROP's unique structure poses various challenges to malware analysis compared to traditional shellcode inspection and detection. Our proposed ROP-based attack vector provides two unique features: 1) the ability to automatically analyze and generate equivalent ROP chains for a given code, and, 2) the ability to reuse legitimate code found in an executable in the form of ROP gadgets. To this end, we have developed a tool named ROPInjector which, given any piece of shellcode and any legitimate executable file, it transforms the shellcode to its ROP equivalent re-using the available code in the executable and finally patches the ROP chain infecting the executable. After trying various combinations of evasion techniques, the results show that ROPInjector can evade nearly and completely all antivirus software employed in the online VirusTotal service, making ROP an effective ingredient for code obfuscation. We believe that this attack vector poses a serious threat which malicious actors can take advantage to perform cyber-attack campaigns. We hope that our work drives more research towards the automatic detection of code obfuscation techniques. The AV industry also should be aware of this code obfuscation technique and counteract accordingly.

1. Introduction

Return Oriented Programming (ROP) gained increased attention during the late 2000's [3] as an advanced stack smashing method to bypass security mechanisms. 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 executable or its loaded libraries each of them ending with a return instruction. These borrowed code chunks are called *gadgets* and their "return" is in fact a call to the next gadget in the chain. As an analogy to regular code, in ROP, gadgets are the "instructions" and esp is the program counter.

Traditionally, the primal usage scenario of ROP has been (and still is) software exploitation to bypass non-executable stack and heap. Other recent works have proposed ROP for other purposes such rootkit development [30], software watermarking [8], steganography [9], and code integrity verification [28]. In this paper, we argue that ROP has another field of application which is code obfuscation and AV evasion. We leverage the fact that ROP is Turing complete and therefore it can perform arbitrary computations; thus every shellcode can be transformed to a ROP equivalent. Our intuition is that ROP's unique structure poses various challenges compared to traditional shellcode detection including verbosity of the gadgets, stack-based chaining, lack of immediates, and the distinction of function calls and regular control flow. On top of that, ROP inherently provides polymorphic properties by selecting a different equivalent gadget for transforming a given instruction or by chaining the selected gadgets in a different order. Another benefit of using ROP for AV evasion is that such borrowed code (that of gadgets) is always benign and tested against false positives. Thus, considering the aforementioned challenges, at its core, ROP can be seen as a new method for code obfuscation.

Based on the above observations, in this paper we present ROPInjector, a tool which, given any piece of code (hereafter, also referred to as shellcode) and any non-packed executable file, it transforms the shellcode into its ROP equivalent and patches it into (i.e. infects) the executable file. ROPInjector [17], which is implemented in C/C++ programming language, infects Portable Executables (PEs) for Windows OS. Since it is very common for AVs to detect minor deviations from the typical arrangement of the file sections and their characteristics (e.g. a second executable section with RWX permissions), besides the transformation of the code into a non-recognizable, non-recurrent form, the developed tool addresses several additional issues to achieve evasion, such as the positioning of the shellcode in the carrier executable and the way of transferring control to the shellcode. Moreover, we have performed several experiments to evaluate the effectiveness of the proposed tool by injecting shellcodes to well-known executable files including acrobat reader, firefox, etc. The high evasion results show that ROPInjector combined with simple behavioral anti-profiling techniques can render AV ineffective and incapable to detect ROP-based malware.

The rest of the paper is organized as follows. Section 2 presents and compares with ROPInjector the related work. Section 3 elaborates on the architecture of the proposed ROPInjector and its functionality details. In section 4, we analyze experimental results and in section 5 we provide a discussion of possible mitigation techniques and the challenges of ROPInjector. Finally, section 6 concludes the article.

2. Related Work

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 alternative uses of ROP. More specifically, in [8] the authors propose ROP for benign purposes; specifically, they use 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. Towards this direction, RopSteg [9] 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 to hide selected code protection by generating equivalent ROP gadgets and blending them into the executable. Both works [8], [9] are reminiscent of ROPInjector as they take advantage of ROP to hide a message or a piece of code inside a PE. However, none of these two solutions evaluate the detection capabilities of ROP, as their main intention is the use of ROP for benign purposes such as software watermarking and steganography. In addition, these solutions do not release a tool to further evaluate their capabilities. Finally, ROPOB [10] is a code transformation technique to obfuscate control flow 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 easily discover the real control flow of the program. However, the main limitation of ROPOB is that through dynamic analysis the obfuscation trivially breaks.

A similar work to ROPInjector is presented in [23], in which the authors leverage ROP to hide malicious code in a PE file. The novel idea is to encrypt the ROP chain to avoid detection and remotely trigger the ROP execution at the runtime. Although this idea is interesting, the encryption itself can raise suspicions, due to high entropy, while a proper key management solution always requires extra effort. Nonetheless, the solution described in [23] presents yet another interesting use case of ROP for malicious purposes. The work closest to ours is presented by Mohan et al. [15]. The authors have 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 entirely randomly selected binaries residing in a system to avoid signature matching detection. They deduce 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 evaluate them semantically with the given malicious instructions. Finally, it performs the suitable code arrangements to construct the final payload. Frankenstein differs to our proposed ROPInjector in several aspects. First, Frankenstein considers a relaxed version of a gadget. That is, while in the classic definition of ROP [20], a gadget ends with the *ret* instruction, Frankenstein definition of a gadget is any sequence of bytes that are interpretable as valid x86 instructions. On the contrary, ROPInjector relies on the pure and original definition of ROP gadgets. Second, Frankenstein main purpose is to modify only code snippets which may look suspicious, while ROPInjector transforms the whole shellcode to the ROP counterpart. Third, authors have only implemented a proof of concept (which is not available publicly).

Finally, we examine two non-academic works that have the same purpose with ROPInjector; that is, to infect PE files with common (possibly encrypted) shellcode in order to bypass AV software.

The first tool named Shellter [1], focuses on maintaining the original structure of the PE file, by avoiding injection of the shellcode into predefined locations or changing the characteristics of the existing sections. It achieves so by overwriting existing code for which it is certain that will be given control during execution of the program. The latter is deduced by tracing the executable file and analyzing its execution flow. Shellter is also capable of reusing imports of the original PE file to change the writing permissions of the section containing the shellcode so that encrypted and self-modifying code can be used. It is also capable of injecting "junk code" before the shellcode that delays execution as a means to anti-emulation. Shellter is advanced in terms of dynamically selecting the location of the patch in the shellcode (as opposed to extending the text section). However, while it features a patching method that introduces variability (as to where in the file is the shellcode injected), it relies on traditional polymorphism methods, that are still subject to signature generation and detection of write permissions or modifications of the text section in memory. Moreover, our proposed approach can introduce variability too, due to the transformation to ROP (which is dependent on the PE file).

The second tool is PEinject [2], which is mostly a method (and referenced as such) rather than a full-featured tool. It injects the shellcode in the (first sufficiently large) padding space found in the text section (either 0xCC caves or section padding) and does not encode or modify the payload in any way, neither does it anticipate for self-modifying or encrypted payloads. Control is passed to the injected shellcode by modifying the address of entry point of the PE file's NT_HEADER.

A common limitation of the aforementioned related works is the limited applicability and practicability of their proposed techniques. For instance, [23] discuss only theoretically the proposed solution, while [9] presents a proof of concept which has been manually prepared only for evaluation purposes. On the other hand, [15] applies the proposed techniques only to a very limited number of code snippets. In contrast to all other previous works, ROPInjector is the first tool released as open source that exploits the polymorphic properties of ROP for advanced code obfuscation. ROPInjector has been designed to be generic and scalable in the sense that given as an input any functional shellcode and any legitimate executable, it will produce an executable that will retain its legitimate functionality bundled with the malicious shellcode. Finally, ROPInjector is fully automated and no manual work is required.

3. ROPInjector

3.1. High-level overview

ROPInjector takes as an input a benign PE and a malicious piece of code (i.e., shellcode) and outputs the PE infected with the malicious piece of code in a ROP form (see figure 2). In general, the ROPInjector approach can be divided into 5 distinct phases as follows:

mov esp, ebp *pop ebp* *ret(n)* *CCCCCCCCCCCCCCCCCC*	*jmp epilogue; normal flow avoiding gadget* *mov ecx, eax; the injected gadget* *jmp return; gadget flow avoiding std. epilogue* epilogue*:* *mov esp, ebp* *pop ebp* return*:* *ret(n)* *CCCCCCCC*

Fig. 1. *Injection of gadget (right) in 0xCC cave preceded by standard function epilogue (left)*

1. Reverse analysis of machine code
2. Finding ROP Gadgets in PE
3. ROP Compilation and Injecting Gadgets
4. Chaining Gadgets
5. Passing Control to the Shellcode

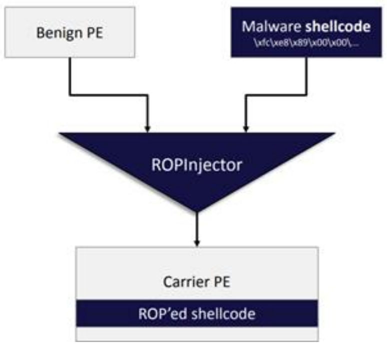

Fig. 2*. ROPInjector high level functionality*

These phases are executed sequentially one after the other. If any of the above phases fail to complete, then ROPInjector outputs an error message describing the cause of the failure. In the next sections, we are going to analyse each of the above phases providing detailed examples to gain a better understanding of the presented notions.

3.2. Reverse analysis of machine code

Reverse analysis of machine code into data structures that are easy to handle is crucial to perform any kind of patching, modifications, re-assembly, and any transformation to ROP. Two are the most important pieces of information required: i) the origin and destination of all relative references (e.g. a relative jump and its target) and ii) which registers are being written or read during each instruction, as well as which registers are free to modify. The former is required for injecting or removing instructions from a code segment without breaking its functionality. The latter is particularly useful to enhance gadget matching, either by performing permutations, or by using gadgets that contain redundant but safe instructions (in this case, unsafe are branch, privileged, or indirect addressing mode instructions because they risk raising errors such as access violation).

3.2.1. MOD/REG/RM and SIB unrolling: Instructions using the MOD/REG/RM indirect addressing mode with displacement or the Scaled Index Byte (SIB) addressing scheme in the shellcode are treated specially before the transformation to ROP. Such instructions are unwanted for the following reasons:

1. They are long (in the best and not so likely case 3 bytes long: 1 for opcode, 1 for MOD/REG/RM and 1 for SIB) hence unlikely to be found in gadgets;
2. They often read many general purpose registers at once, thus reserving them while as mentioned earlier, the more the free registers the better;
3. Their respective gadgets (should they be found or injected) will probably not be reusable, due to the use of displacement and index constants (e.g. *mov edx, [esi*2+16]*).

In order to circumvent this kind of situations, we reduce such instructions to their arithmetic equivalents one-by-one. We call this process *unrolling* and it is performed to the shellcode before any transformation to ROP. For instance, *mov eax, [ebx+ecx*2]* may be replaced by:

[a'] *Mov eax, ecx*
[b'] *sal eax, 1*
[c'] *add eax, ebx*
[d'] *mov eax, [eax]*

If the register *eax* is not free to use for the arithmetic operations, another temporary register that is free may be used.

Noteworthy is how unrolling unlocks register access from one atomic instruction to many. For instance, in the example, *ecx* is freed at [a'] and *ebx* at [c']. If for example *eax* were to be freed at the preceding 10 instructions, then instructions [a'] to [c'] could be moved 10 instructions behind, thus resulting in an additional free register (i.e., *ecx* and *ebx*, but not *eax* which will not be free) in that preceding code chunk.

3.3. Finding gadgets

Candidate gadgets in the executable sections of the given PE file must end in one of the following instructions:

- *ret,*
- *retn,*
- *pop regX,*
- *jmp regX, or jmp regX.*

Exceptionally for the latter, the gadget in question must be first paired with a *loader gadget* that loads the required return address into *regX*. The process begins by finding all gadget endings and temporarily storing them to a list. For each of those endings, *n* bytes of preceding machine code is disassembled for each *n* up to maximum depth *N* (typically 20 bytes). If such disassembly aligns with the ending (not guaranteed since x86 instructions are of variable length) a candidate gadget has been found. Candidate gadgets containing any illegal, privileged (e.g. *sysenter, int, iret*), branch or *esp* modifying instructions are filtered out.

3.4. ROP compilation and injecting gadgets

The gadgets found in the aforementioned process are first analyzed instruction-by-instruction to infer register access. Since gadgets are allowed to contain safe but redundant instructions, their register access is tested for modifications to the register in question (e.g. *mov ecx, eax; pop ecx; ret;* gadget cannot be used for moving *eax* to *ecx*) as well as the non-free registers of the source instruction to be encoded.

Following that, they are lifted into an Intermediate Representation (IR) consisting of an operation-type, and 3 operands with different meaning depending on the type. If a multi-instruction gadget contains more than one representable instructions, only the first is considered. However, the following ones have also been considered in other gadgets with the same ending, because of the backwards gadget finding process described in the previous paragraph. Noteworthy is the fact that by parsing into this higher level IR, one-to-one permutations are automatically performed. That is because both gadgets and instructions are classified into one of these types, based on which the encoding is then performed, rather than on the instructions per se. The IR is also useful for selecting the encoder function accompanying every gadget. Encoders are responsible to answer "*whether their assigned gadget can encode a given instruction*", as well as to encode it into a list of stack operations if requested to.

Predefined, one-to-one permutations (i.e. one instruction to one gadget) are achieved through the IR and encoder functions. Encoders will also perform basic algebraic permutations based on the properties of addition, subtraction multiplication and division. For instance, if the instruction to be encoded is of type ADD_IMM (*add reg, imm*), an encoder will repeat anything *add reg, x* with *x* being an integer divisor of *imm*, *imm/x* times. Addition and subtraction with constants will also be swapped if the signs of the constants are flipped. M-to-N permutations quickly scale to exponentially growing space and are out of the scope of this work.

In order to enhance transformation of the source shellcode, and since not all required gadgets are always found in the PE file, new ones are also injected as needed. Firstly, the 0xCC caves (i.e., padding commonly left by the linker in-between code segments in the text section of PEs) are used for this injection, and if they are filled, the text section is extended before the actual patch. The injection is performed in the least noticeable way to avoid alarms. If a standard epilogue (*mov esp, ebp; pop ebp; ret*) is found right before the 0xCC cave, the gadget is injected in-between the preceding code and the epilogue. Figure 1 depicts such an example gadget injection of a *mov ecx, eax* gadget. In the case that no epilogue is found at the boundary with the 0xCC cave, a pseudo-function with standard prologue and epilogue is injected to avoid heuristics or n-grams that might raise suspicion due to non-ordinary returns. This pseudo-function has the following form shown in Figure 3.

```
        push ebp
        mov ebp, esp
        <gadget code>
        jmp return
        mov esp, ebp
        pop ebp
return:
        ret
```

Fig. 3. *Pseudo-function ending used during gadget injection*

Algorithm 1 as shown in figure 4 is used when ROPInjector builds the ROP equivalent of a given shellcode. The first step of the algorithm is to scan the PE and find all the pieces of code that could potentially be used for chaining to the provided shellcode, named gadget endings. As a last step of the initialization, the algorithm locates the 0xCC nests (also known as caves) where gadgets can be generated and injected if not found in the PE. In case one or more suitable gadgets are found for the instruction, then the best one is chosen. In this scope, best gadget is considered to be the one that contains less instructions. Next, all instructions are checked against the list of candidate gadgets. We have to distinguish one important case here: in case an instruction cannot be encoded by any of the gadgets found in the previous step, then a suitable gadget needs to be generated and injected (see line 10 of algorithm 1). For this purpose, the 0xCC nests are considered as presented in algorithm 2 (see figure 5). That is, if there is space, then the needed gadget is injected in the caves. On other hand, if the gadget cannot fit, then the text section of the PE is extended to create more space.

```
Algorithm 1 ROP compilation
Input: shellcode, PE
Output: gadgets_list
 1: count gadget endings
 2: find CCnests in PE
 3: i ← first instruction of shellcode
 4: while not end of instructions do
 5:     if i can't be ROP-compiled then
 6:         continue
 7:     else
 8:         gadget ← best matching gadget
 9:         if best gadget not found in PE then
10:             gadget ← inject gadget in CC nests
                // Described in Algorithm 2
11:         end if
12:     end if
13:     if gadget not targeted by jump then
14:         chain to the previous gadget
15:     else
16:         set as new gadget chain
17:     end if
18:     append gadget to gadgets_list
19:     i ← next instruction
20: end while
21: return gadgets_list
```

Fig. 4. *ROP Compilation algorithm*

```
Algorithm 2 Injecting gadgets in CC nests
Input: instruction
Output: gadget
gadget ← assemble new gadget for instruction
cc_nest ← find space for gadget
if not enough space then
    cc_nest ← make space from padding or extending .TEXT
end if
if cc_nest is a new gadget section then
    write function prologue
    inject gadget
    write function epilogue
else
    add gadget to existing section
end if
return gadget
```

Fig. 5. *Gadget injection algorithm*

Continuing now with algorithm 1, gadgets either injected or found, are used to encode each one of the (allowed) instructions contained in the shellcode. The last step of this algorithm concerns the chaining of the gadgets (either found or injected). For this part we have to distinguish two cases:

1. If a gadget is targeted by a branch instruction, then the algorithm handles it as a new chain where more gadget chains can be added.
2. If the gadget is not targeted, the algorithm just adds it to an existing gadget chain.

3.5. Chaining Gadgets

The return address chain can be built either during runtime or during compile-time and saved to the initialized data section of the file (to be then copied at runtime to the stack). The most alarming option would be the first (during runtime) and we choose this to evaluate our evasion ratio. During this process, besides the pushing of the VAs onto the stack, the ROP compiler must consider pushing immediate constants, adjustments for stack pointer modifications in the gadget (e.g. redundant *pops*, *retns*) and gadgets with loader gadgets. For this purpose, the following types of *stack operations* are defined:

PUSH_VA	; push a (loader) gadget VA onto the stack
PUSH_IMM	; push an immediate constant onto the stack
ADVANCE	; advance (subtract from) the stack pointer a number of bytes
CHAIN	; pseudo operation denoting a placeholder for the next gadget's VA

The result of the encoding process of a given instruction by a given gadget is a series of stack operations for the invocation of the gadget. The list of such operations for all gadget calls describes the assembly instructions that if executed, will build the chain in the stack. Alternatively, such operations may be used to create the required stack frame during compile-time, save it as initialized data and copy it over from the data section during runtime. The latter process allows also for encoding/decoding of the stack frame. In the former case, and when multiple calls are made to the same gadget (e.g. as in using *inc eax* to achieve *add eax, 3*) the compiler wraps the call with a conditional jump loop using a free register.

However, not all types of instructions can be easily encoded into ROP. In this work we do not consider the encoding of branches (jumps, calls, loops, interrupts), privileged instructions and pops. Hence, the return-oriented code chunks must finally return back to the source shellcode. This is achieved by wrapping the chain building instructions in the following:

```
[1]    call build_chain
[2]    jmp past_the_chain
[3]    build_chain:
[4]    push <VA of gadget N>
[5]    ....
[6]    push <VA of gadget 1>
[7]    ret
[8]    past_the_chain:
[9]    <other instructions/chains>
```

In this way, the last gadget (N) will return to instruction [2] jumping past the chain building instructions and continuing normal execution flow.

3.6. PE Patching and Passing Control to the Shellcode

First of all, the patching of the PE file and the passing of control to the shellcode must be done in the least noticeable way. A second executable section hosting the shellcode would be too alarming, since the vast majority of executables has only one. The next least disruptive and easy to implement option would be to inject the shellcode once again in the 0xCC nests. However, there may not always be sufficient space in those 0xCC caves, as ROPInjector puts this padding space to inject the required missing gadgets.

For the above reasons we choose to append the shellcode to the existing text section of the executable, and correct all section headers and relocations accordingly. To pass control to it, the default practice is to replace the instructions pointed to by NT_HEADER.AddressOfEntryPoint with a jump to the shellcode, which is appended to those replaced instructions followed by a jump back to the original execution flow. Directly pointing the address of entry point to the shellcode in this case is avoided, since many AVs' heuristics are alarmed by the fact that it points towards the end of text section. An alternative to giving control to the shellcode at program entry, is to hook any calls to ExitProcess, exit or other similar functions. This technique in particular, as shown also later by the results, bypasses behavioral profiling by AVs that employ emulation or sandboxing. This can be attributed to the fact that either AVs emulate only a small portion of the executable's entry code due to scanning time constraints, or because of a lack of techniques for triggering a graceful exit (i.e., many programs do not handle SIGINT and SIGTERM signals).

4. Experiments and Results

4.1. Methodology

In this section, we present the methodology adopted to evaluate ROPInejctor. We have selected for carrier PEs 9 popular 32-bit executables (see first column of Table 1). Regarding the source shellcode, we selected two of the most

Table 1 List of PE files used as carriers in the experiments and various statistics (ROP Entry method and reverse tcp shell)

PE	Size (KB)	File Version	Candidate Gadgets Found in PE	Gadgets Injected	Gadgets Used From PE
Acrobat.exe	650	19.10.20069.49826	338	78	12
AcroRd32.exe	1423	11.0.8.4	5599	67	26
cmd.exe	305	6.3.9600.16384	608	76	16
firefox.exe	439	63.0.3.6892	1634	65	28
java.exe	187	8.0.192.12	1148	76	18
nam.exe	1828	1.0a11a	3515	67	25
notepad++.exe	2783	7.6.0.0	7778	55	41
Rainmeter.exe	39	2.4.0.1678	11	83	0
wmplayer.exe	163	12.0.9600.19145	60	82	3

popular payloads of Metasploit framework [5]: i) (staged) meterpreter reverse tcp, and, ii) the reverse tcp shell. For each PE (9 in total) and each shellcode (2 in total) we performed 4 patching scenarios as analyzed in Table 2: i) ROP Entry ii) ROP Exit iii) Shellcode Entry iv) Shellcode Exit. Thus, we created 72 different samples (i.e., 72 infected PEs). On these samples we derived the following results:

1. Various statistics including: a) number of candidate gadgets found in the PEs, b) number of gadgets re-used from the PEs, c) missing gadgets that were injected, d) number of instructions of the 2 tested shellcodes that were transformed to ROP equivalents.
2. Similarity measurements between original PE and infected PE.
3. ROPInjector Code complexity metrics.
4. Evasion results for each PE and top 10 popular antivirus detection results.
5. Evasion results for EMET.

Note that all conducted experiments were performed on a Windows 7 32-bit operating system.

4.2. Statistics and similarity measurements

Statistics and information regarding the infected PE are generated for each execution of ROPInjector using the method ROP Entry and as shellcode the reverse tcp shell (see Table 1). As expected the largest executable (i.e., notepad++.exe) includes the highest number of candidate gadgets (i.e., 7778). However, only 41 gadgets were used, while 55 were generated and injected by ROPInjector. On the opposite side, in the smallest executable named Rainmeter.exe, ROPInjector discovered only 11 candidate candidates and none of them were eventually used to build the ROP chain (i.e., all gadgets for Rainmeter.exe were generated and injected by ROPInjector - 83 in total).

We also derived statistics for the shellcodes that we tested. In particular, we examined how many instructions of the shellcode, ROPInjector actually transformed them to ROP equivalents. In particular, the reverse tcp shell consists of 193 instructions out of which 139 were replaced with gadgets by ROPInjector, while (staged) meterpreter reverse tcp includes 187 instructions and 129 were transformed eventually to ROP equivalents.

In the next experiment, we derived the similarity between the benign PE and the infected PE. The aim of this experiment was to measure quantitatively the modifications that ROPInjector causes to the original PE. We used as a metric the cosine similarity, since it has been used by previous works to detect obfuscations of malware variants [29]. We tested 3 different PEs taken from table 1 and as a shellcode we selected reverse tcp shell. The direction we followed, was to analyze the text section of the two PEs and compare them as strings. Table 3 shows the numerical results of our experiments (in our case cosine similarity takes value from 0 to 1, where 1 means exact matching). As shown in table 3, in all methods of ROP transformation (ROP entry or ROP exit) the similarity was very high (~0.99 in all cases), while for patching methods without ROP transformation (Shellcode entry and Shellcode exit) the similarity was even equal to 1 (for java.exe). This outcome is a direct result of ROPInjector approach to re-use code found in the carrier PE introducing only some minor modifications to the PE. In essence, the main modification by ROPInjector occurs when it resorts to the injection of missing gadgets in the code caves, only when the carrier PE does not provide the appropriate gadgets for the ROP transformation of the shellcode.

Table 2 List of patching methods tested

Patch Method	Description
ROP Entry	The executable file is patched with the shellcode unrolled, converted to ROP and the entry point before the original PE code.
ROP Exit	The executable file is patched with the shellcode unrolled, converted to ROP and the entry point before the original program’s exit (hook ExitProcess or exit).
Shellcode Entry (no-rop no-unroll)	The executable file is patched with the shellcode intact and the entry point before the original PE code.
Shellcode Exit (no-rop no-unroll)	The executable file is patched with the shellcode intact and the entry point before the original program’s exit (hook ExitProcess or exit).

Table 3 Cosine Similarity between original PE and infected (reverse tcp shell)

PE	ROP Entry	ROP Exit	Shellcode Entry	Shellcode Exit
Dummy	0.9989297	0.9989285	0.9999999	1.0
firefox	0.9999774	0.9999777	0.9999913	0.9999908
java	0.9984519	0.9984519	1.0	1.0
notepad++	0.9992218	0.9992218	1.0000000	1.0000000

4.3.Code complexity

ROPInjector has a large codebase and analyzing the entire program to calculate a unique complexity value is not feasible. To this end, we have analyzed the complexity of the most important functions of ROPInjector using various software metrics. In particular, we have estimated: i) Lines of Code (LOC), ii) Cyclomatic complexity which is a quantitative measure of the number of linearly independent paths through a program's source code, and, iii) Number of input parameters. As shown in table 4, the most complex and large function of ROPInjector was classifyGadget() (cyclomatic complexity equal to 330) followed by classifyInstruction() (cyclomatic complexity equal to 199). A question that naturally arises is "What is the functionality of these functions?". The purpose of classifyGadget() function is to arrange the candidate gadgets, discovered in the legitimate PE, into categories based on the type of instructions they could potentially replace. Similarly, classifyInstruction() encodes the instructions of the shellcode into categories of the intermediate representation. Due to their complex functionality, these functions share the largest LOC and cyclomatic complexity.

4.4. Antivirus evasion

In order to evaluate ROPInjector against antivirus detection, we used the VirusTotal online antivirus scanning service [4], which at the time of writing the paper included 57 AV engines.

Table 4. Software metrics for ROPInjector code complexity

Function name	LOC	Cyclomatic Complexity	Number of Parameters
classifyGadget()	692	330	1
classifyInstruction()	354	199	1
ropCompile()	301	85	9
makeRel32Branch()	114	28	3
patchPEInMemory()	107	19	3
analyze()	40	6	4

For each PE and each shellcode we performed our 4 patching scenarios (as listed in Table 2). Figure 6 depicts the evasion ratios ($1 - \frac{detection}{total\ \#\ of\ AVs}$) of ROPInjector for each one of the four scenarios for the Meterpreter reverse tcp shellcode. We can observe that the executables generated by the ROPInjector in the "ROP Exit" scenario achieved the highest evasion ratio. In particular, in more than half of the test cases. the ROPInjector results almost in 100% AV evasion. This means that in average ROPInjector achieves AV evasion equal to **98.31%,** as depicted in figure 7 (i.e., "ROP-Exit" scenario).

From these results we can deduce that evasion depends almost equally on both code obfuscation/transformation (hence signature evasion) and entry point (hence behavioral profiling evasion). This can be attributed to the fact that some AVs were able to detect ROPInjector despite the fact that there is no signature, due to the ROP polymorphism. It seems that behavioral analysis is equally important to static signatures for some AVs (from the ones that were alarmed) and is mostly performed during entry of executables, while the exit of the executables is not monitored equally.

We have also derived the detection effectiveness of the top 10 most popular antivirus engines [31] against the ROPInjector for the ROP exit method. We tested AVs against 9 infected PEs with the meterpreter reverse tcp. In table 5, the numeric values next to the engine name refer to the number of files that were detected. An interesting observation is that many of the popular commercial solutions used widely in organizations fail to score high in the detection of the infected PEs. For instance, McAfee failed to detect any of the infected executables, while AVG detected only one infected PE. Kaspersky and ZoneAlarm had the best performance as they detected 5 out of 9 PEs.

Table 5. Top 10 Antivirus detection results (reverse tcp shell and ROP Exit)

AV	Detection results
Kaspersky	5/9
ZoneAlarm	5/9
BitDefender	1/9
AVG	1/9
McAfee	0/9
Webroot	0/9
TrendMicro-	0/9
Sophos	0/9
ESET-NOD32	0/9
Panda	0/9

4.5. EMET evasion

Apart from AVs, we evaluated the evasion effectiveness of ROPInjector against a sophisticated anti-malware tool named Enhanced Mitigation Experience Toolkit (EMET). The latter is developed by Microsoft aiming to mitigate exploitation attacks including ROP which are leveraged by malware. During the runtime of an executable, 50 security-wise critical functions are monitored by EMET (i.e., hooked) and several security checks are performed before a monitored critical function is allowed to be executed. The most important and relevant to ROP security checks of EMET are briefly discussed below:

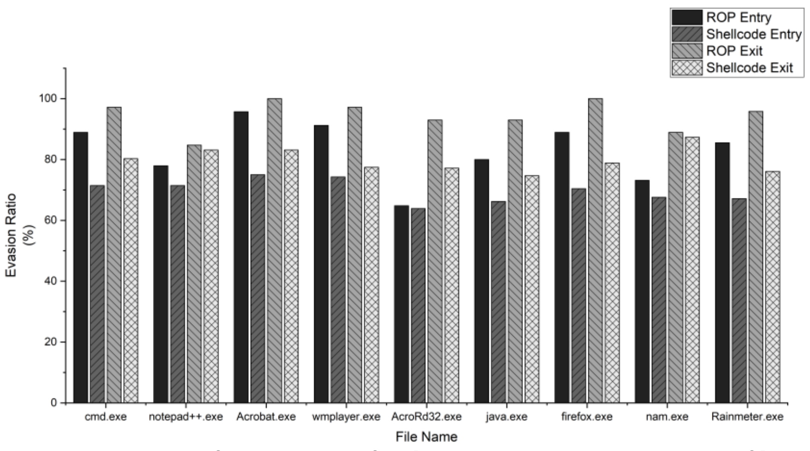

Fig. 6*: Evasion ratio of ROPInjector for the meterpreter reverse tcp per file*

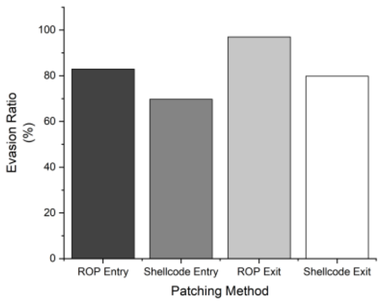

Fig. 7*: Evasion ratio of ROPInjector for the reverse meterpreter payload per method (ROP-exit scenario)*

- **Caller Check** rule ensures that when a critical function is invoked, it is reached via a Call instruction rather than a Ret.
- **SimExecFlow** is the reverse of Caller Check rule. After a call to a critical function is completed, this protection simulates the execution to ensure that the following code is legitimate (and not a ROP chain). The first return address is given on the stack, and subsequent return addresses are deduced by simulating instructions that modify the stack/frame pointer. The SimExecFlow rule checks that each return address is preceded by a Call instruction to appear legitimate.
- **StackPivot**. Upon entering a critical function, EMET checks to ensure that the stack pointer register is within the thread's upper and lower specified stack limit. This guards against pivoting the stack pointer to attacker-controlled memory such as heap.
- **MemProt** check is triggered when a specific critical function named VirtualProtect() is called, in order to check whether the attacker is changing execution permissions on stack memory pages.
- **LoadLibrary** rule ensures that a shellcode doesn’t use remote file paths by invoking the critical function LoadLibrary().

If any of these security checks detects a ROP based attack, EMET closes the application and creates an appropriate log event.

To evaluate the evasion effectiveness of ROPInjector against EMET detection capabilities, we scanned the ROPInjector infected executables against EMET (version 5.5). As we observed in our experiments, EMET techniques proved to be ineffective against ROPInjector as it was not able to detect any executable generated by ROPInjector. This can be attributed to the fact that the exploitation scenarios that EMET tries to protect are different from the exploitation scenarios of ROPInjector. More specifically, the aforementioned EMET rules aim at protecting against malware that try to exploit a vulnerability and perform a memory corruption attacks (i.e., buffer overflows). In such scenarios, a malware leverage ROP attacks after a buffer overflow, in order to call specific functions which would further extend its capabilities. On the other hand, ROPInjector does not rely on memory corruption attacks for its nefarious purposes; instead ROPInjector stores (i.e., backdoors) a malicious piece of code (i.e., shellcode) in a ROP based form inside a legitimate executable. When the legitimate executable is executed (using for example social engineering or HID attacks), the backdoored shellcode is also executed.

To exemplify, we consider the MemProt rule and how it fails to protect against ROPInjectgor. A common goal of malware is to enable a memory region to be executable, in order to execute subsequently a malicious shellcode. This is typically achieved by performing a buffer overflow to pass the control to a ROP chain, in order to call the VirtualProtect() function. EMET’s MemProt rule can protect against such attacks as it monitors the VirtualProtect() function, but not against ROPInjector, since the latter neither has to call VirutalProtect() nor it has to perform memory corruption attacks as the whole executable is under its control. As a side note, it is important to mention that various research works such as the one presented in [27] have shown that EMET rules can be bypassed trivially and in general they are insufficient to protect against memory corruption attacks.

5. Discussion

Detection. Most antivirus software relies on string signatures and mild behavioral profiling detection mechanisms. By encoding the shellcode into its return-oriented equivalent and even by performing elementary mutations (unrolling), the former can be bypassed in the vast majority of cases. Of course, the return address chain has to be built somehow in the stack and that would leave a footprint subject to signing. The process involves either pushing the return addresses to the stack or just copying the whole chain from another memory location (possibly some .data segment) and adjusting the stack pointer. However, a) the kind of code required for such operations is very common and seemingly benign, b) it largely depends on the attacked PE and its image base since in the worst case it is a series of `push <VAi>` operations, and c) can be randomized or encrypted in many and trivial ways. Especially, what we mentioned for reason (b) holds because gadget addresses change for different PEs and different image bases, hence changing the footprint and statistics of the chain building instructions even if they are for the same source shellcode. Dynamic analysis and behavioral profiling can also be avoided by carefully intercepting normal execution flow in points that AVs either cannot emulate or simply cannot derive enough evidence to classify the behavior as malicious. In this work, we presented as a means to the latter the hooking of common calls to process exit, resulting in many cases in absolute evasion and in others rates greater than 98%. However, if the anti-dynamic analysis protection of ROPInjector is bypassed than the dynamic analysis of an infected PE can reveal the malicious code hidden in a ROP form. However, we consider this as out of scope since ROPinjector is mainly designed as an obfuscation tool to avoid static detection. Advanced techniques specialized for anti-dynamic analysis such as malwash [16] can be combined and further enhance evasion capabilities of ROPInjector.

The techniques presented can still be mitigated if dealt with individually. For instance, signatures could be created for ROP building instructions and behavioral analysis could be also performed backwards in terms of process life-cycle. However, since slight variations and randomization can again disarm scanners, a more robust countermeasure does not seem straight-forward to design, and/or practical to implement.

Several AVs rely on code statistics (such as entropy, n-grams and more), in order to classify PE files as benign or malicious. Unless such methods are designed to consider separate parts of the file's code, ROPInjector does not affect the statistic metrics of the binary file as a whole. The only metric that is affected and can be used by AVs for detection is the hash of the PE itself. In particular, for well-known PEs (such as Firefox.exe), AVs can cross-check the hash of the PE. If the hash is different from the official version of the executable, then AVs can raise an alarm. However, since new versions of these applications are continuously released, maintaining the legitimate hash values of each PE is a challenging task.

Control flow integrity and whitelisting. A mechanism that counteracts against ROP attacks is Control Flow Integrity (CFI) [12]. The latter is a compile time mechanism for preventing malicious code from redirecting the execution flow of a program. In other words, the goal of CFI is to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution. This prevents code-reuse techniques such as ROP from working because they would cause the program to execute control-flow transfers which are illegal under CFI [13]. By its own definition, we can deduce that ROPInjector does not violate CFI policies, since the purpose of CFI is to defend against threats which will modify *the runtime operation* of an executable. On the contrary, ROPInjector change the execution flow of an executable *in the compile time* and specifically, changes the entry or exit point of the executable in order to transfer the code execution into the ROP chain.

Perhaps the most promising direction is towards the strict coupling of the host operating system with the trusted software certificates (or checksums) and a "default distrust all". policy, i.e., whitelisting rather than blacklisting, which may have gain in popularity the last years in Windows domain environments (i.e., application whitelisting [6]), but not in standalone installations of Windows OS in personal environments.

Hardware Performance Counters. Modern processors (such as Intel, AMD, ARM) support a variety of hardware performance counters (HPC) [22] for monitoring and measuring events during process execution related to instructions (e.g., instruction retired, cycles), memory accesses (e.g., cache hits or misses, main memory hits or misses), and the execution behavior on the CPU pipeline, among others. Although performance counters were initially introduced for debugging purposes, recent research has shown that HPC can be utilized for anomaly detection caused my security incidents including software exploitation attacks [19], [25] and malware detection [24]. More specifically, a work that is designed for ROP detection based on HPC is named SIGDROP [25], in which the ROP detection is based on the assumption that for a given number of return misses, the number of instructions retired is lower than some threshold. An extension of SIGDROP is ROPSentry [19], in which the authors leverage several hardware events occurred by a ROP based attack, such as instruction translation lookaside buffer (ITLB) misses, last level cache (LLC) misses, mispredicted return instructions and number of return instructions.

In general, we argue that the above techniques could raise the bar for our approach. However, a crucial aspect that should be taken into account when evaluating ROPInjector against these countermeasures is our freedom to craft the PE in a way that can make these defenses irrelevant. That is, ROPInjector backdoors a shellcode inside a legitimate executable in a ROP form not only by re-using the ROP gadgets of the executable but also by generating and injecting new gadgets. In this way, ROPInjector can construct the ROP execution flow in such a way that it can fly under the radar of the above techniques. A prime example is presented in [22], in which the authors deceive the HPC based detection indicators of SIGDROP. That is, the authors exploit the effects of non-determinism and overcounting, which are inherent in HPC, in order to deliberately add noise and evade SIGDROP detection.

Generality and Scalability. ROPInjector has been designed to be generic and scalable in the sense that given as an input any functional shellcode and any legitimate executable, it will produce a poisoned executable that will retain its legitimate functionality bundled with the malicious shellcode. However, it is important to mention that we have considered only shellcode and PE files which are compiled for the x86 architecture. There is no provision for other architectures such as x64 or ARM. Moreover, ROPInjector is not designed to analyze other types of Windows executables (e.g., batch files) or executables for other platforms such as Linux (e.g., elf format), since all manipulations of the executable are done with the assumption that the provided binary is in PE format. A more technical remark here is that there is a defined set of instructions that cannot be compiled to a ROP representation. Specifically, all instructions accessing ESP register such as push and pop instructions are not replaced by ROP equivalents. ROPInjector leaves these instructions untouched and continues with the subsequent instructions. From an implementation point of view, ROPInjector depends on the output of the ProView disassembler which is a C library. As such there is the possibility that ROPInjector fails to generate the poisoned executable, in case of code ambiguity in the ProView disassembler output. Finally, ROPInjector has been implemented in C/C++ programming language, making it difficult to add new features and extensions. A huge improvement towards this direction would be to port ROPInjector to a higher level object oriented language (e.g. Python).

Difficulty and Limitations. Here we analyze the difficulties to generate ROP equivalent shellcodes for a given PE. In ROPInjector each instruction could potentially be replaced by one or more ROP gadgets using different permutations. But in order to limit the difficulty of searching in the generated space by one-to-many permutations, we decided to limit the search to one-to-one permutations (i.e., one instruction to one gadget). In order to achieve one-to-many permutations ROPInjector should additionally implement software synthesis and verification methodologies [11], [26], which is out of the scope of the tool. Another difficulty of ROPInjector is related to the fact that ROPInjector must ensure that the execution of the original legitimate program will not be affected to avoid raising suspicions. To this end, ROPInjector performs extra verifications to ensure that a gadget does not write to undesired memory locations of the original executable memory space, fact that could have unknown side effects (e.g., trigger a segmentation fault). Finally, it is important to mention that the lack of free registers when encoding an instruction to its ROP equivalent, is one of the biggest hurdles that ROPInjector can encounter. That is, a candidate ROP gadget can be selected only if it clobbers registers which their values are not required by the rest of the instructions. Thus, ROPInjector must keep track of registers state (i.e., free or not free). In case ROPInjector cannot find such a gadget because there are not enough free registers, then it does not convert the instruction to a ROP equivalent.

6. Conclusions

In this paper we presented ROPInjector, a software tool which, given any piece of shellcode and any non-packed 32-bit Portable Executable (PE) file, it transforms the shellcode to its ROP equivalent and patches it into (i.e. infects) the PE file. The main outcome of this research is the developed algorithms for: a) analysis and manipulation of assembly code on the x86 instruction set, and b) the automatic chaining of gadgets by ROPInjector to form safe, and functional ROP code that is equivalent to a given shellcode. Experiments showed that ROPInjector can transform a detectable shellcode to an undetectable ROP form. We hope that our work drives more research towards the automatic detection of code obfuscation techniques. The AV industry also should be aware of this code obfuscation technique and counteract accordingly.

7. References

[1] Shellter project, https://www.shellterproject.com, last accessed on April 15, 2018

[2] Injecting Shellcode into a Portable Executable(PE) using Python, http://www.debasish.in/2013/06/injecting-shellcode-into-portable.html, last accessed on April 15, 2018

[3] Shacham, Hovav. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007.

[4] VirusTotal, https://www.virustotal.com, last accessed on June 15, 2018

[5] Metasploit, http://www.metasploit.com/, last accessed on April 15, 2018

[6] Applocker, https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview, last accessed on February 15, 2018

[7] Experimental Windows .text section Patch Detector, https://github.com/nccgroup/WindowsPatchDetector, last accessed on July 15, 2018

[8] Ma, Haoyu, et al. "Software Watermarking using Return-Oriented Programming." Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. ACM, 2015.

[9] Lu, Kangjie, Siyang Xiong, and Debin Gao. "Ropsteg: program steganography with return oriented programming." Proceedings of the 4th ACM conference on Data and application security and privacy. ACM, 2014.

[10] Dongliang Mu et al. "ROPOB: Obfuscating Binary Code via Return Oriented Programming." International Conference on Security and Privacy in Communication Systems. Springer, Cham, 2017

[11] Jiang Ming et al. "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking." Proceedings of the 26th USENIX Security Symposium. 2017.

[12] Martín Abadi et al. "Control-flow integrity." Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005

[13] Nathan Burow et al. "Control-flow integrity: Precision, security, and performance." ACM Computing Surveys (CSUR) 50.1 (2017): 16.

[14] Shacham, H. et al. "Return-oriented programming: Exploits without code injection." Black Hat USA Briefings, August 2008.

[15] Vishwath Mohan and Kevin W. Hamlen. "Frankenstein: Stitching Malware from Benign Binaries." USENIX Workshop on Offensive Technologies (WOOT 12012): 77-84.

[16] Kyriakos Ispoglou, and Mathias Payer. "malWASH: Washing Malware to Evade Dynamic Analysis." WOOT. 2016.

[17] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. "Ropinjector: Using return oriented

programming for polymorphism and antivirus evasion."Blackhat USA (2015).

[18] Giorgos Poulios, “Advanced Antiviurs Evasion Techniques”, Master Thesis, Department of Digital Systems, University of Piraeus, 2015.

[19] Das, Sanjeev, et al. "ROPSentry: Runtime defense against ROP attacks using hardware performance counters." Computers & Security 73 (2018): 374-388

[20] Roemer, Ryan, et al. "Return-oriented programming: Systems, languages, and applications." ACM Transactions on Information and System Security (TISSEC) 15.1 (2012): 2

[21] Ruan, Yefeng, Sivapriya Kalyanasundaram, and Xukai Zou. "Survey of return‐ oriented programming defense mechanisms." Security and Communication Networks 9.10 (2016): 1247-1265

[22] S. Das, J. Werner, M. Antonakakis, M. Polychronakis and F. Monrose, "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security," 2019 IEEE Symposium on Security & Privacy (SP), San Fransisco, CA, US, pp. 345-363.

[23] Borrello, Pietro, Emilio Coppa, Daniele Cono D’Elia, and Camil Demetrescu, “The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse”, 34th ACM/SIGAPP Symposium On Applied Computing. Limassol, Cyprus, April 2019.

[24] Tang, Adrian, Simha Sethumadhavan, and Salvatore J. Stolfo. "Unsupervised anomaly-based malware detection using hardware features."International Workshop on Recent Advances in Intrusion Detection. Springer, Cham, 2014.

[25] Wang, X. & Backer, J. SIGDROP: Signature-based ROP Detection using Hardware Performance Counters. arXiv preprint arXiv:1609.02667 (2016).

[26] Blazytko, Tim, et al. “Syntia: Synthesizing the semantics of obfuscated code.” 26th USENIX Security Symposium, 2017.

[27] DeMott, Jared. “Bypassing EMET 4.1.”, IEEE Security & Privacy 13.4 (2015): 66-72.

[28] Andriesse, Dennis, Herbert Bos, and Asia Slowinska. “Parallax: Implicit code integrity verification using return-oriented programming”, 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2015.

[29] Karnik, Abhishek, Suchandra Goswami, and Ratan Guha. “Detecting obfuscated viruses using cosine similarity analysis.” First Asia International Conference on Modelling & Simulation (AMS’ 07). IEEE, 2007.

[30] Vogl, Sebastian, et al. "Persistent Data-only Malware: Function Hooks without Code." NDSS. 2014.

[31] The Best Antivirus Protection for 2019 https://www.pcmag.com/roundup/256703/the-best-antivirus-protection