University Of Piraeus

Department of Digital Systems

Postgraduate Programme "Digital Systems Security"

Master's Thesis

Analysis and Evaluation of ROPInjector

Tsioutsias Theodoros, MTE1633

Under the supervision of:
Dr. Christoforos Dadoyan, dadoyan@unipi.gr

Piraeus 2019

This thesis is dedicated to my family and especially to Z&M.
Many thanks to Dado for all the help and support.

Table of Contents

Table of Figures

Table of Tables

Abstract

This thesis is to analyze the shellcode injecting tool named ROPInjector. The tool explores the potential of Return Oriented Programming as an antivirus evasion technique. The analysis is going to be twofold. Firstly, we are going to present the functionality of the tool by diving into the various algorithms used for transforming malicious code into its ROP equivalent. We are going to present the purpose of each mechanism and discuss about the implementation. At the same time, we will perform a detailed qualitative inquiry by comparing the input and the resulting (i.e. infected) binaries in order to evaluate the performance and effectiveness of ROPInjector. Last but not least, using the results of this analysis we are going to propose next steps in order to make the tool friendlier to the user, allowing researchers of the community to use it for their work.

1 Motivation

The main motivation for this thesis is to broaden our knowledge regarding Return Oriented Programming (ROP) as a technique for antivirus evasion. Especially, we are interested in the various approaches used to inject transformed (i.e. ROP compiled) malicious code in binary executable files. To effectively accomplish this, we chose to investigate and further took apart the inner mechanisms of ROPInjector [1]. This thesis is to analyze in depth and present the purpose of the different algorithms used for infecting Portable Executables (PEs) [2] as well as the procedure for the ROP transformation of the given shellcode.

We are also going to evaluate the performance of the tool, analyzing its behavior and experimenting with various infecting targets and shellcodes. Lastly, we are going to propose future work that can be done related to ROPInjector.

2 Return Oriented Programming

2.1 ROP against DEP

The most common way for attackers to exploit memory corruptions in programs was to point its flow of execution to malicious code written in the stack. From the mid 2000s, operating systems, started providing protection against stack smashing attacks. The way the stack was protected was to mark the memory space where data is written as non-executable. This measure of protection made it almost impossible to exploit memory corruptions using the “traditional” ways. The mechanism is called Data Execution Prevention (or DEP).

Return-oriented programming was presented as a technique to overcome the new means of protection. In this technique for exploiting memory errors, the attacker can take over the execution flow of the targeted program by borrowing and arranging chunks of already existing code in a specific order. These borrowed chunks of code are in fact small instruction sequences ending with a “return” (i.e. RET or 0xC3) instruction, hence return-oriented, are called gadgets. The gadgets have to end with the RET instruction in order to allow the chaining of multiple such sequences.

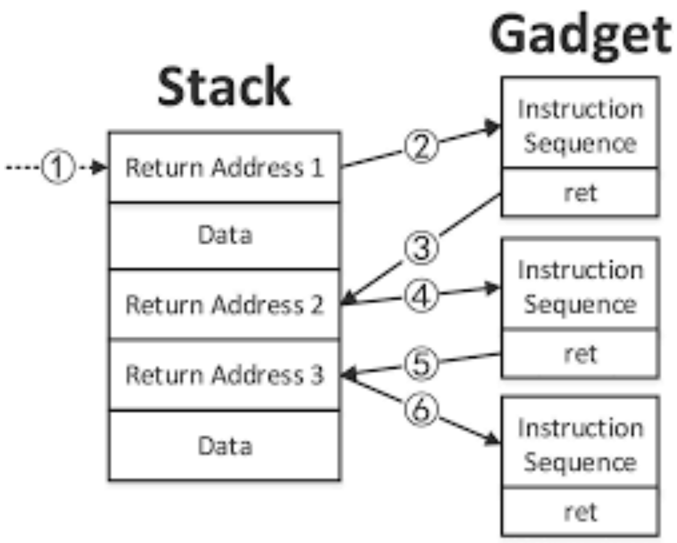


Figure 1. Return Oriented Programming Example

Chaining ROP gadgets can be accomplished by carefully pushing into the stack the virtual addresses of the already existing gadgets. This means that an attacker can assemble a working shellcode without injecting malicious code into the targeted executable.

2.2 Antivirus evasion

Antiviruses are the key defense and protection against the installation and execution of malicious software. Current antiviruses provide multiple security features in order to detect and disable malwares. They can be categorized in two main areas:

1. **Signature based detection.** In this mode antiviruses work by scanning the system for signatures and then testing them against a database containing the signatures of known viruses.
2. **Heuristic based detection.** This mode of operation has to do with the real-time protection of the system. Antiviruses try to detect programs performing suspicious tasks like setting up network connectivity etc.

2.2.1 Polymorphism for AV evasion

Polymorphism is used to evade signature based and pattern matching detection utilized by antivirus software. Polymorphic malware is a type of malware that changes its "identity" to avoid detection by such security solutions. Many forms of malware are known to be able to be polymorphic including viruses, worms, trojans etc.

A polymorphic malware will keep its functional purpose unchanged and at the same time it will change its identifiable characteristics. Characteristics that can be changed include from file names and file types to encryption keys. The purpose of these transformations is to generate a new signature every time the program is executed so that is unrecognizable by antiviruses.

To perform this kind of transformations, polymorphic malwares require to have a writeable code section that comes with the executable or change its permissions at execution time. Both of these characteristics are identifiable and alarming to the majority of Antiviruses.

2.2.1.1 Packers

This usually is short for "runtime packers" which are also known as "self-extracting archives". Software that unpacks itself in memory when the "packed file" is executed. Sometimes this technique is also called "executable compression". This type of compression was invented to make files smaller. So, users wouldn't have to unpack them manually before they could be executed. But given the current size of portable media and internet speeds, the need for smaller files is not that urgent anymore. So, when you see some packers being used nowadays, it is almost always for malicious purposes. In essence to make reverse engineering more difficult, with the added benefit of a smaller footprint on the infected machine.

2.2.1.2 Crypters

The most usual technique used by crypters is called obfuscation. Obfuscation is also used often in scripts, like javascripts and vbscripts. But most of the time these are not very hard to bypass or de-obfuscate. More complex methods use actual encryption. Most crypters do not only encrypt the file, but the crypter software offers the user many other options to make the hidden executable as hard to detect by security vendors as possible the same is true for some packers. Another thing you will find in that post is the expression FUD (Fully Undetectable) which is the ultimate goal for malware authors. Being able to go undetected by any security vendor is the holy grail for malware authors. But if they can go undetected for a while and then easily change their files again once they are detected, they will settle for that.

2.2.2 ROP for AV evasion

Although Return-Oriented Programming was introduced as an attack against Data Execution Prevention mechanisms, its ability to produce functionality from already existing code, proves its potential as an antivirus evasion technique by enabling polymorphism.

With ROP, the attack is performed using seemingly benign pieces of code that already exist in the targeted executable and, therefore, already are tested against antiviruses. Furthermore, the whole procedure of building the chain of gadgets is a series of push <VA> operations or even copying the same gadgets from other locations. Considering also, the fact that gadgets are dependent on the attacked executable, in different targeted executables the same shellcode will result in different signatures.

Although the procedure of gadget chaining can be subjected to signing, the main advantage ROP is that it enables polymorphism without requiring a writeable code section.

3 ROPInjector

ROPInjector is a tool applying in practice exactly what was described above. The tool is written in Win32 C and its purpose is to infect any 32-bit Portable Executable given a shellcode. By transforming the shellcode into its return-oriented equivalent and patching the targeted PE, it successfully proves that ROP can be used as an alternative to polymorphism for antivirus evasion.

In this chapter, we will try to present a high-level description of the tool's functionality. Its functionality can be broken down in two main functions:

- transformation of the given shellcode
- patching the transformed shellcode into the targeted PE.

3.1 Shellcode Transformation

3.1.1 Preparation

In order to prepare the given shellcode, ROPInjector, firstly performs a reverse analysis on it. The main purpose of this analysis is to parse the shellcode in an intermediate representation that will enable the tool's next steps. This intermediate representation consists of predefined data structures that hold useful information regarding for the ROP compilation. At this step, the first transformations occur. Firstly, the relative branches are processed and translated to 32-bit equivalent instructions where needed in order to avoid possible overflows during patching. The second transformation has to do with the elimination instructions using indirect addressing mode or the SIB addressing scheme. These instructions are transformed because they are long and not usually found in gadgets.

3.1.2 ROP compilation

In this step, ROPInjector composes a return-oriented equivalent of the source shellcode. To begin the procedure, the tool finds all the potentially usable gadgets in the targeted binary. By locating all the suitable instructions for gadget chaining and filtering out gadgets containing unusable instructions, the list of candidate gadgets is created. Following this procedure all these gadgets are parsed in higher-level intermediate representation as well, based on their meaning and expressing the instructions they could potentially encode. In the case where more gadgets are needed then new ones are crafted and injected in the 0xCC nests of the targeted binary. To ensure the successful matching of the discovered (or injected) gadgets with the shellcode, the tool will apply elementary one-to-one permutations on the source code. The last segment of this procedure is to create the chain of used gadgets by defining the stack operations that have to occur in order to ensure the execution of the shellcode and finally returning to the attacked program.

3.2 PE patching

The purpose of this function is to inject the malicious code as well as the code allowing its execution in the targeted PE. The code could be injected in the 0xCC nests in the PE but, as mentioned before, they are used in order to inject the ROP gadgets that weren't available in the .TEXT section of the PE. Another option would be to create a second executable section in the binary, but this would be obvious for antiviruses to pick up. The choice the authors of the tool made was to append the shellcode in the existing .TEXT section readjusting the rest of the sections accordingly. The tool provides two options for passing control to the shellcode. The shellcode can be linked to the entry of the PE or its exit. In the first option, the shellcode is executed before passing the control back to the program. This is done by changing the NT_HEADER.AddressOfEntryPoint and pointing it to the first instruction of the shellcode (or the first push VA) at the end of the shellcode there is jump to the previously set entry point. In the latter option, all the calls to EXIT_PROCESS are replaced with a jump to the shellcode instead.

ROPInjector, also, except for making the control transfer from the infected program to the shellcode in the least noticeable way, it attempts to apply some more techniques in order to achieve lower detection ratios. The first of them, is adding a delay (i.e. call to sleep) before the execution of the shellcode. After patching a signed PE and of course recalculating its checksum, ROPInjector hides its certificates by deleting the pointer to them.

4 ROPInjector Algorithms

In this section we will be analyzing some of the most important algorithms of ROPInjector. The approach we chose to follow is to firstly translate them in pseudocode in order to be able to analyze their purpose and functionality.

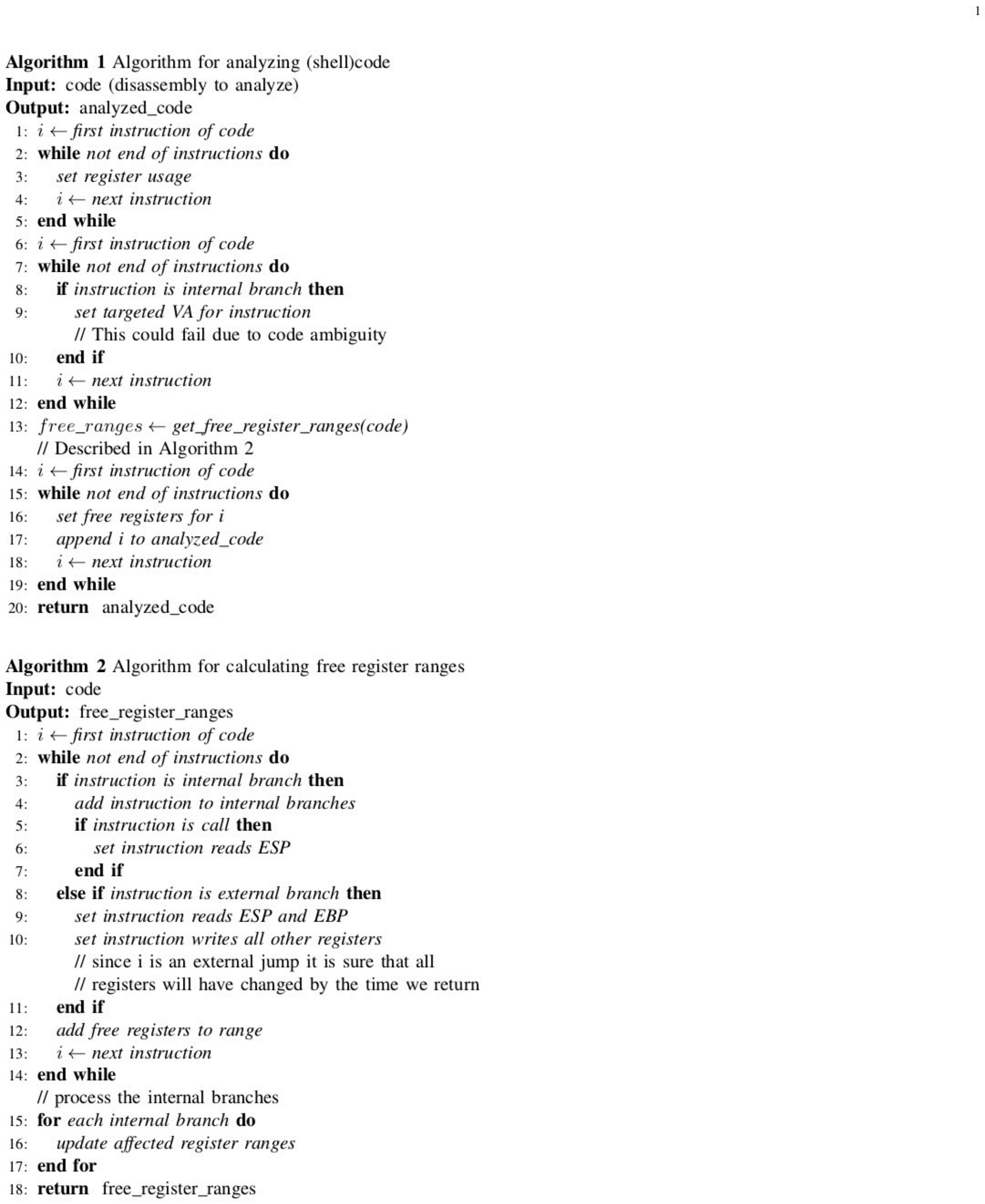
4.1 Algorithm for analyzing disassembly

This algorithm is used multiple times throughout the execution. Its main purpose is to transform the output of the disassembler into the data structures that ROPInjector requires as well as to initialize them with information gathered about each instruction. In some cases, the algorithm is also used to verify the validity of a disassembly.

As a first step, the aforementioned data structures are initialized. Looping over each disassembled instruction and after a set of static checks, it sets the used registers for this instruction. As a second step, all the branch instructions that have a target virtual address within the provided segment of the disassembled code, are marked as internal branches. For each internal branch the targeted VA is also set into the instruction structure.

The next step is a bit tricky, and for this reason it is treated as a separate algorithm. This is the part of the code analysis that calculates the instruction ranges where registers are free. Free register in this scope, means that changing the value of the register, would not affect the execution of the analyzed program. A free range for a register, from the time the stored value is last read (marked as read) to the point where a new value is stored again (marked as written) again. So, this bit decides which registers are safe to use for other purposes for each instruction(s) that we might need to transform. This is done mainly to enable the ROP transformation of the shellcode.

To do that, all the branch instructions are checked. Specifically, all internal branches are stored for later use and in the case that the instruction is a “call” (internally), then ESP is set as being read. For external branches (branching to a VA outside the provided segment) all ESP and EBP are read and all other registers are considered as written. The registers are considered as written, because there is no way to know how they are being treated outside the segment that is being analyzed. Taking into account all the internal branches store in the previous step, the ranges where registers are free to use, have to be adjusted if they are affected.



1

Algorithm 1 Algorithm for analyzing (shell)code
Input: code (disassembly to analyze)
Output: analyzed_code

```
$i \leftarrow$ first instruction of code
while not end of instructions do
    set register usage
    $i \leftarrow$ next instruction
end while
$i \leftarrow$ first instruction of code
while not end of instructions do
    if instruction is internal branch then
        set targeted VA for instruction
        // This could fail due to code ambiguity
    end if
    $i \leftarrow$ next instruction
end while
$free\_ranges \leftarrow$ get_free_register_ranges(code)
// Described in Algorithm 2
$i \leftarrow$ first instruction of code
while not end of instructions do
    set free registers for i
    append i to analyzed_code
    $i \leftarrow$ next instruction
end while
return analyzed_code
```

Algorithm 2 Algorithm for calculating free register ranges
Input: code
Output: free_register_ranges

```
$i \leftarrow$ first instruction of code
while not end of instructions do
    if instruction is internal branch then
        add instruction to internal branches
        if instruction is call then
            set instruction reads ESP
        end if
    else if instruction is external branch then
        set instruction reads ESP and EBP
        set instruction writes all other registers
        // since i is an external jump it is sure that all
        // registers will have changed by the time we return
    end if
    add free registers to range
    $i \leftarrow$ next instruction
end while
// process the internal branches
for each internal branch do
    update affected register ranges
end for
return free_register_ranges
```

Figure 2. Algorithm for analyzing disassembly

4.2 Algorithm for ROP compilation

This algorithm is used while ROPInjector tries to build 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. From those, filters out the ones that cannot be used (mostly because of containing illegal instructions) as well as duplicates. As a last step of the initialization, the algorithm locates the 0xCC nests(/caves) where gadgets can be injected if not found.

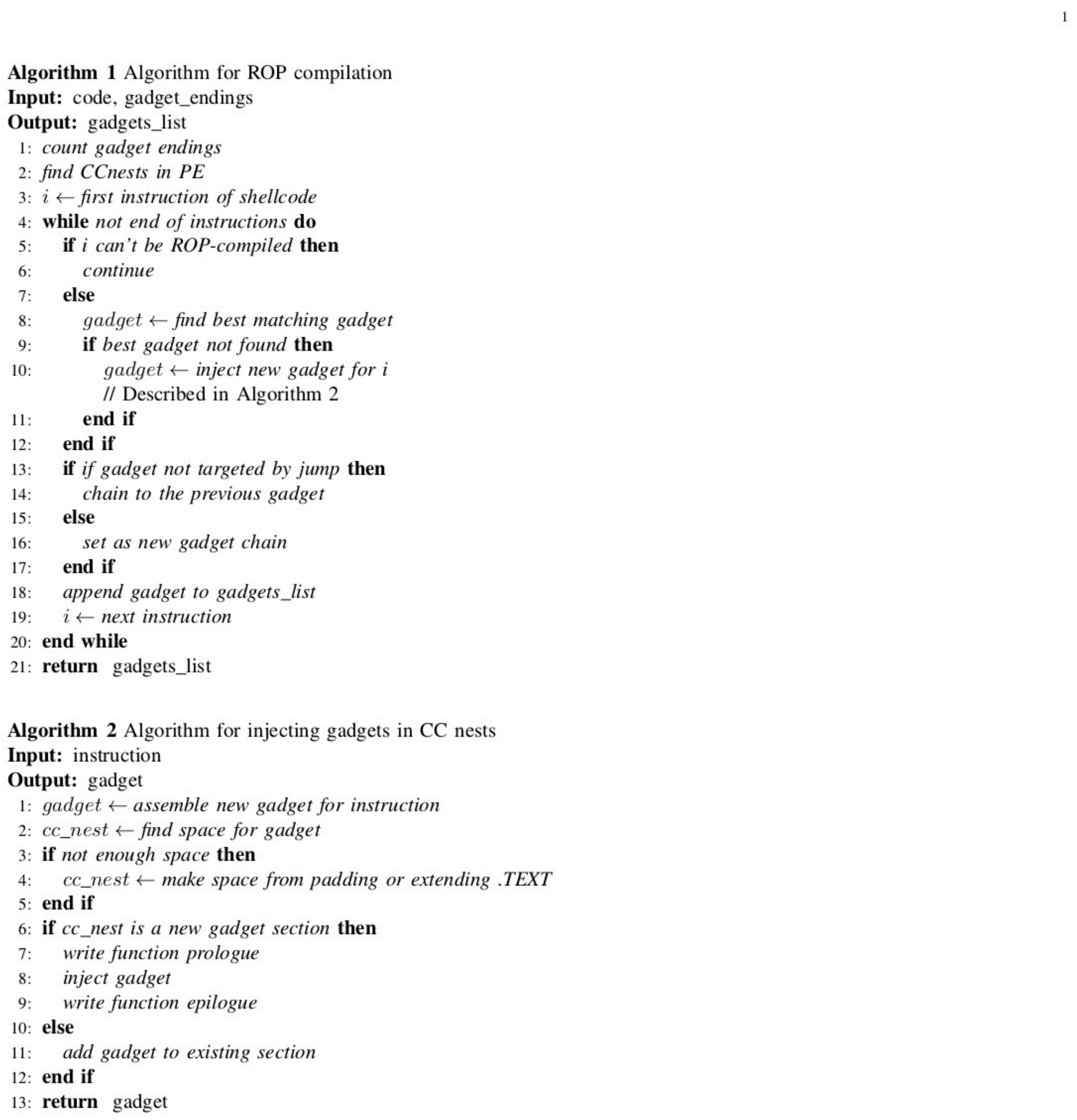
At this point, all instructions are checked against the list of candidate gadgets. We have to distinguish two cases here:

- In case an instruction cannot be encoded by any of the gadgets found in the previous step, then a suitable gadget needs to be injected. For this purpose, the 0xCC nests are used. If there is space, then the needed gadget is injected there. If the gadget cannot fit, then the .TEXT section of the PE is extended to create more space (more 0xCC nests).
- 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 and will result in less VAs in the gadget chain.

Either injected or found, gadgets 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). Again, for this part we have to distinguish to cases:

- If a gadget is targeted by a branch instruction, then handle as a new chain where more gadget chains can be added.
- If the gadget is not targeted just add it to an existing gadget chain



```
Algorithm 1 Algorithm for ROP compilation
Input: code, gadget_endings
Output: gadgets_list
count gadget endings
find CCnests in PE
$i \leftarrow$ first instruction of shellcode
while not end of instructions do
    if i can't be ROP-compiled then
        continue
    else
        $gadget \leftarrow$ find best matching gadget
        if best gadget not found then
            $gadget \leftarrow$ inject new gadget for i
            // Described in Algorithm 2
        end if
    end if
    if if gadget not targeted by jump then
        chain to the previous gadget
    else
        set as new gadget chain
    end if
    append gadget to gadgets_list
    $i \leftarrow$ next instruction
end while
return gadgets_list
```

```
Algorithm 2 Algorithm for injecting gadgets in CC nests
Input: instruction
Output: gadget
$gadget \leftarrow$ assemble new gadget for instruction
$cc\_nest \leftarrow$ find space for gadget
if not enough space then
    $cc\_nest \leftarrow$ 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
```

Figure 3. Pseudocode for the ROP compiling algorithm

5 Evaluation of ROPInjector

The binaries we will be using throughout the following experiments can be found in the next table:

File Name	File Size (KB)	File Version	Hash (SHA256)
java.exe	187	8.0.192.12	b51c64c7ef4544dd04a76781e8be5b22482e7908b945528 b08c9da73f07b4e4e
cmd.exe	305	6.3.9600.16384	e1a080e61fb1baf0da629d34baee6f0f9d0e0337bf6ced9f4 b3ab9b1c23d91ba
Dummy.exe	411	-	e2e7a86a398900ff65604f0ea391826c22343f0550adb4ba 60285c2e6d2c0e5d
firefox.exe	439	63.0.3.6892	76e344a43910a45679f208f1414bd720ca8efe5ca207d441 79737da30aad090b
EssModel.exe	650	2.2	5b82358f54f21fdb0ea5a39c6d87b2cbfe730af4184e8cd00 43cc1a4e098e6e8
AcroRd32.exe	1423	11.0.8.4	ed820c61c179fa27bb63305b5c18dbe913aea38cecc27835 d3b3e51007e7d575
nam.exe	1828	1.0a11a	5d329bb39ba744cdba5e1afe107551c18ba0acd46cb6764 391024a73aa2d583f
notepad++.exe	2783	7.6.0.0	c517690b5c9a83515b2d6aae6297990fc26ada6f06497507 af714b0f0ea4ee96

Table 1. List of binaries used in the following experiments

Regarding the selected shellcode, we will be using the popular **reverse TCP shell (revshell)** and **reverse TCP meterpreter (revtcp)** of the Metasploit Framework [3].

5.1 Time of execution

In this chapter we will be measuring the times of execution for different modes of operation for ROPInjector. The purpose of this exercise is to understand which individual procedures are the most time consuming. Another useful result, we are hoping to extract from this experiment, is to identify the parameters that can affect ROPInjector's performance.

To achieve that, we will be measuring the overall execution time of ROPInjector as well as the two of its most important individual procedures, shellcode manipulations and PE patching.

In the following charts executables are sorted by size (increasing). For each executable we measured three durations denoted with the following name:

- **Total**
  The overall time of ROPInjector's execution for a given binary

- **Shellcode**
  This measurement shows the time spent by ROPInjector on changing the given shellcode. The procedure includes replacing unwanted instructions with instructions that make following steps easier, the manipulation of ROP gadgets found in the shellcode as well as source code permutations.

- **Patch**
  The time it took ROPInjector to patch the given binary. For different modes of operation, this procedure can include the changes performed in the given binary leading to and actually inserting the given shellcode.

5.1.1 ROP Compile on exit

In this mode ROPInjector compiles the given shellcode trying to create its ROP equivalent and chains its execution on the exit of the target binary.

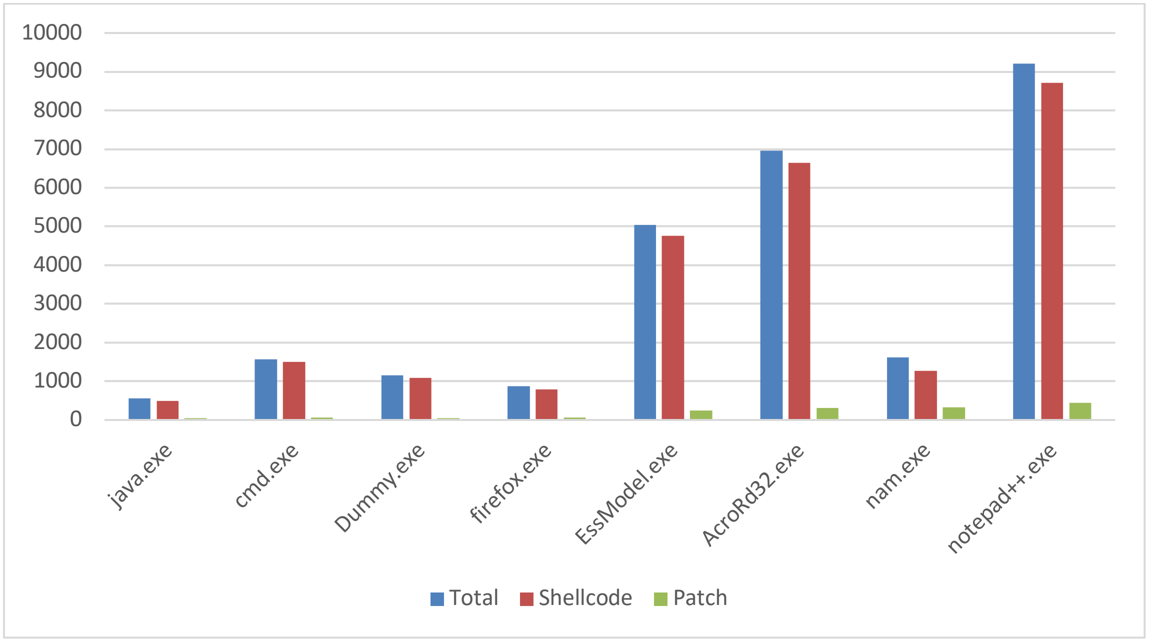


Figure 4. ROP-exit-revshell mode execution times

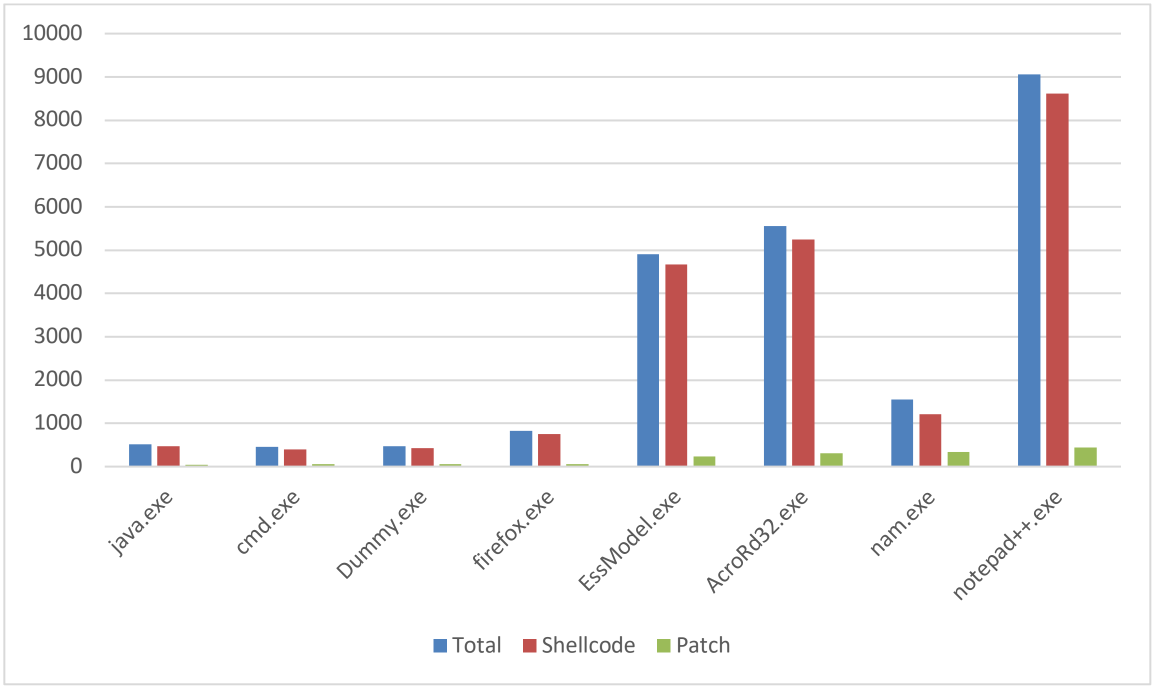


Figure 5. ROP-exit-revtcp mode execution times

As expected, the conversion of the shellcode to its ROP equivalent takes the most time. Patching the PE duration is significantly less. The total execution of the program is also as expected dependent on the size of the given binary.

5.1.2 ROP compile on entry

In this mode ROPInjector compiles the given shellcode trying to create its ROP equivalent and chains its execution on the entry of the target binary.

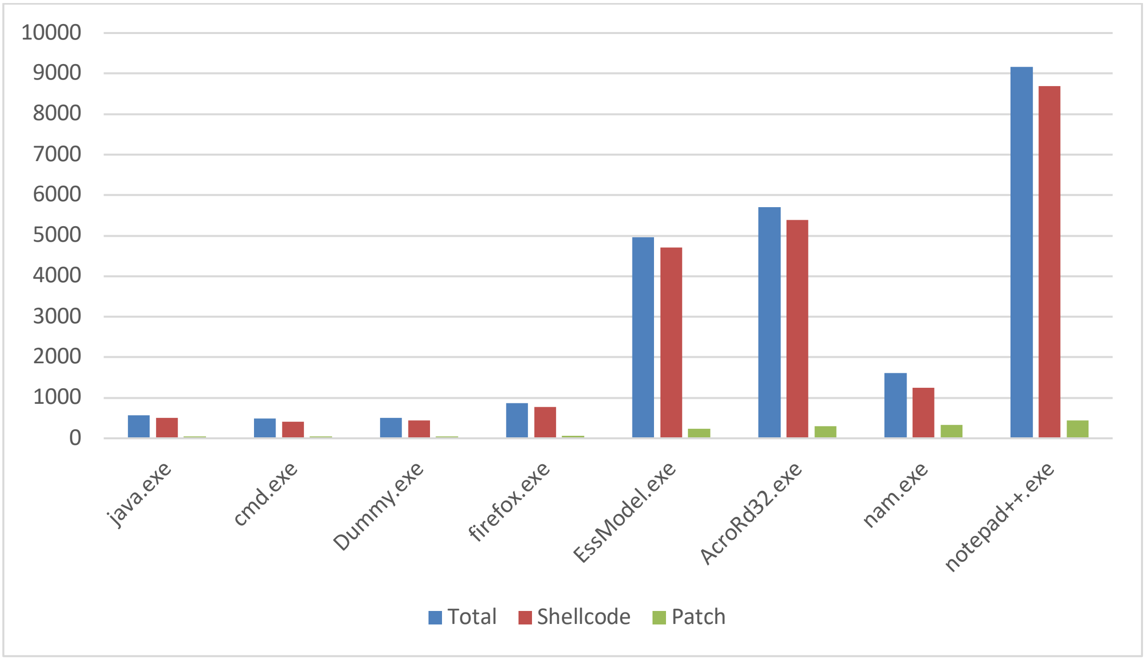


Figure 6. ROP-entry-revshell mode execution times

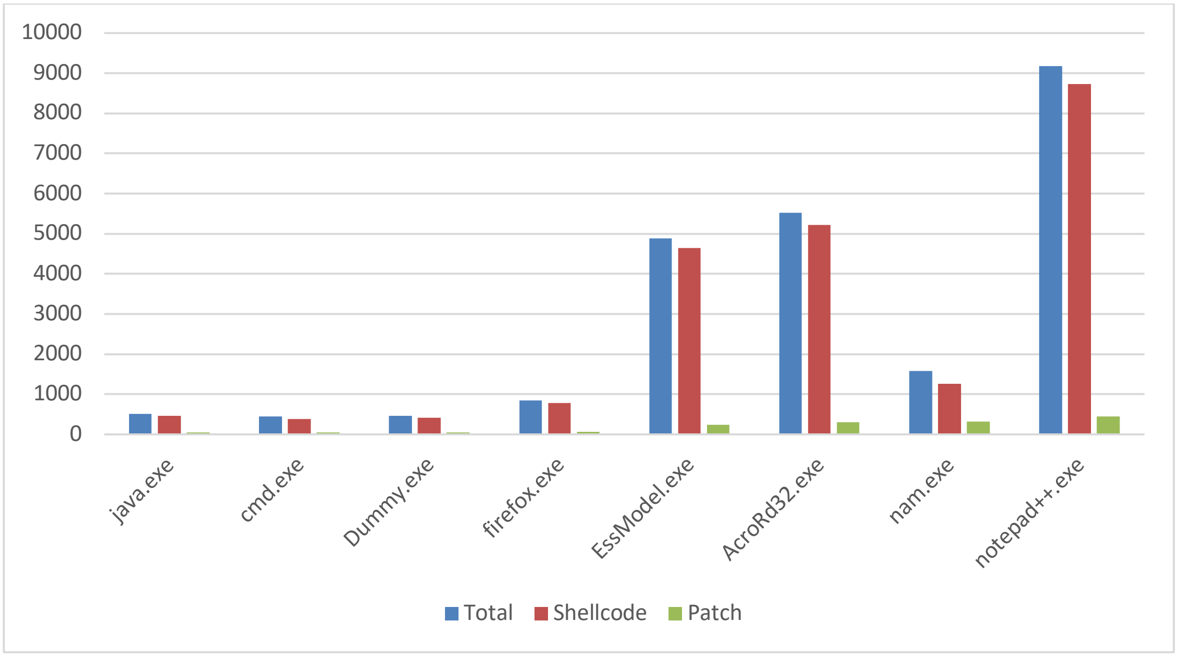


Figure 7. ROP-entry-revtcp mode execution times

There are not many differences with the previous experiment as expected. Modifying the given shellcode is the most time-consuming part of the execution. Patching the binary is almost identical to the previous experiment.

5.1.3 ROP compile on entry with 20 second delay

This mode is the same with the previously presented one, but ROPInjector adds a 20 second delay before the execution of the shellcode.

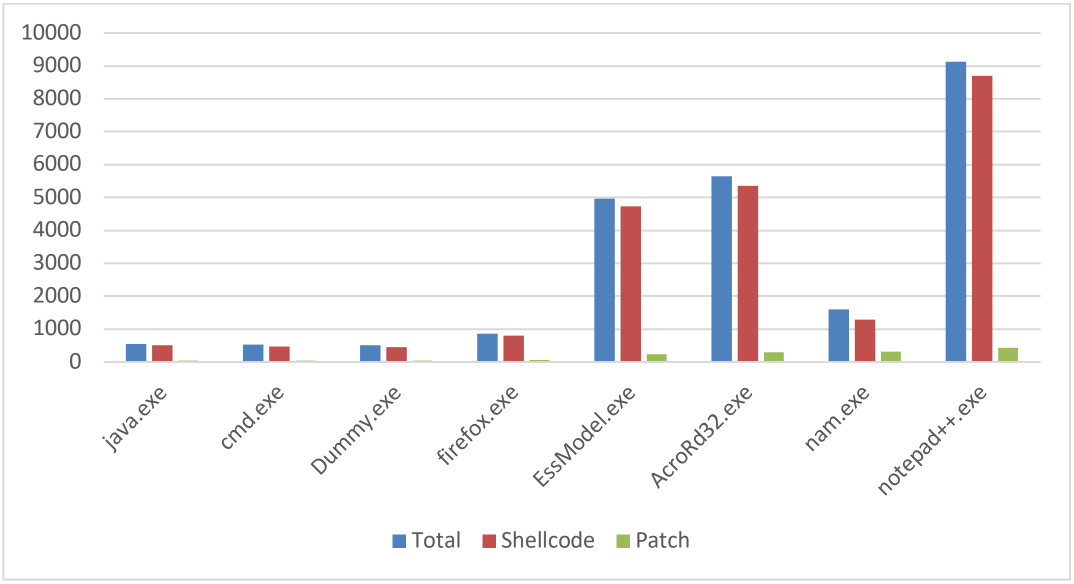


Figure 8. ROP-entry-revshell-d20 execution times

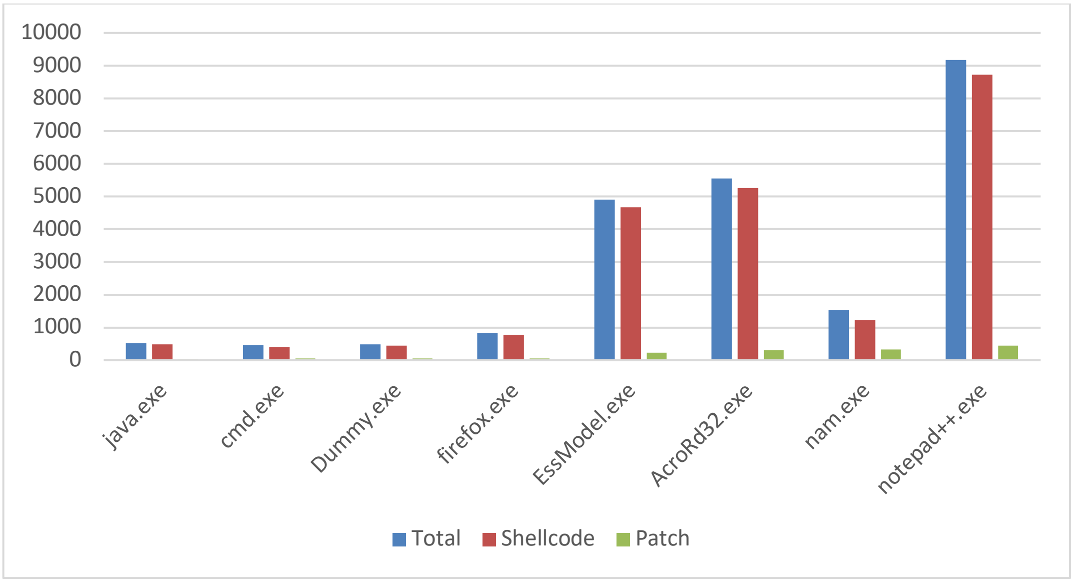


Figure 9. ROP-entry-revtcp-d20 mode execution times

Adding the delay before the shellcode does not seem to affect the execution time.

5.1.4 No ROP compilation on exit

In this mode, the shellcode is not compiled as ROP. ROPInjector, though, applies changes on the given shellcode trying to optimize its matching potential. The main element of these changes is that all the relative branches as well as the Scaled Index Byte (SIB) are unrolled to make easier the procedure of finding their ROP equivalents. Meaning that here, we can easily measure the time in which ROPInjector prepares the shellcode for ROP compilation (without being compiled).

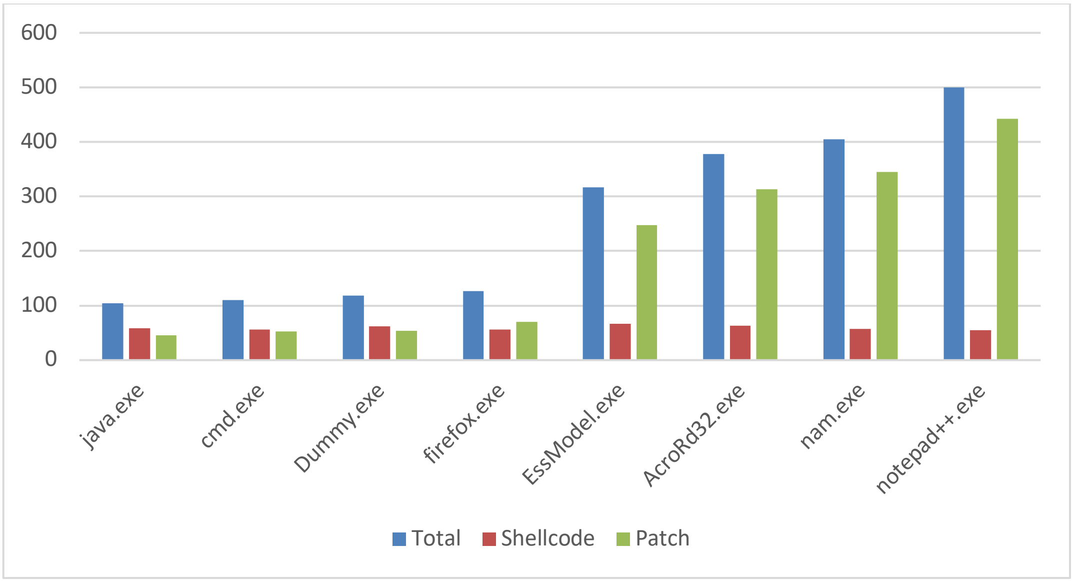


Figure 10. norop-exit-revshell mode execution times

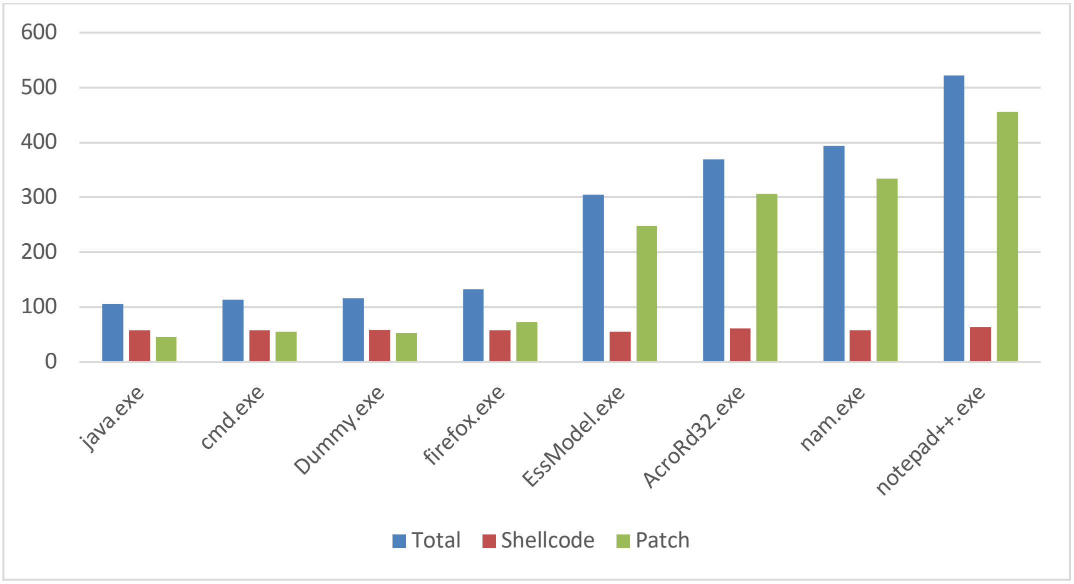


Figure 11. norop-exit-revtcp mode execution times

In this experiment, we can see that the time spent on changing the shellcode is the same for all binaries. This shows that the preparation for the ROP compilation is not depending on the size of the binary, which is expected, since the optimizations applied are totally dependent on the shellcode itself.

5.1.5 Shellcode on entry

In this mode, the shellcode is not compiled as ROP and no changes are applied on it. The given shellcode is chained on the entry of the given binary.

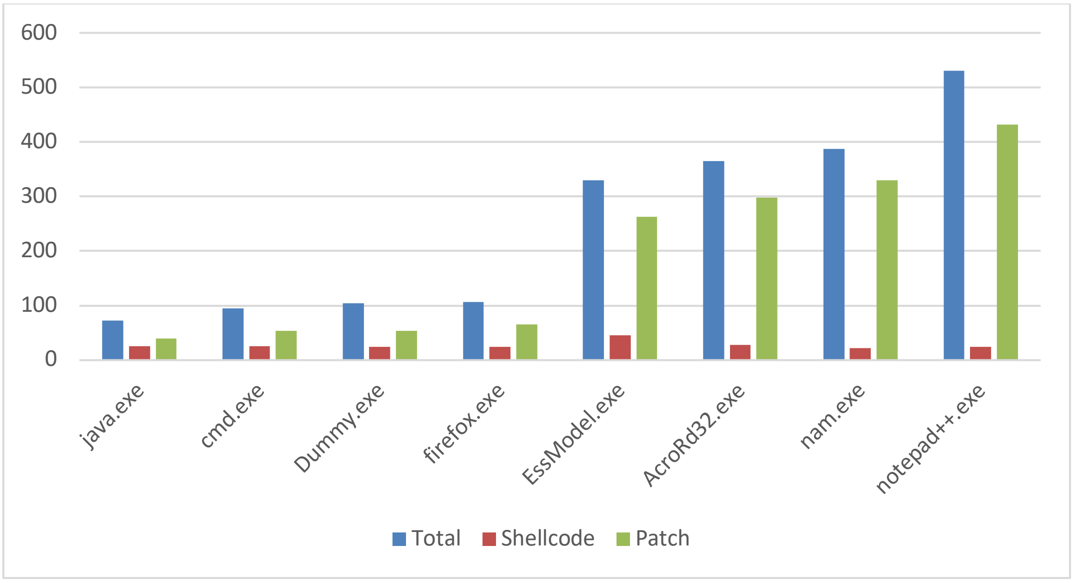


Figure 12. shellcode-revshell mode execution times

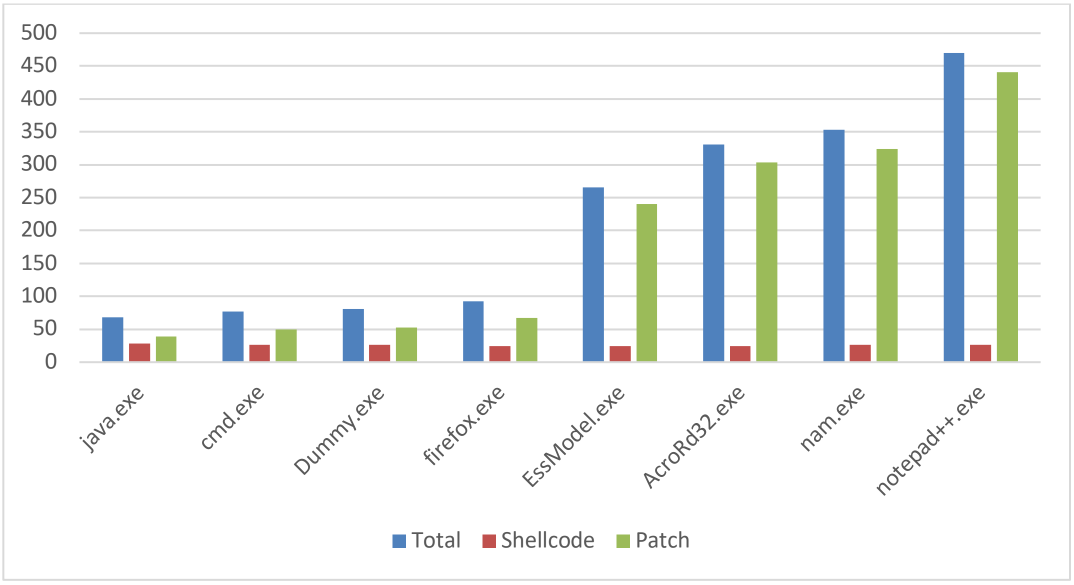


Figure 13. shellcode-revtcp mode execution times

As expected, and since no changes are applied to the shellcode, the time spent changing it is pretty much linear and identical for all the given binaries. Patching the binary, is the most time-consuming part of the part, and is clearly related to the size of the given binary. This can be explained easily, considering that ROPInjector will have to search throughout binary for 0xCC nests, where it can insert the original shellcode.

5.1.6 Shellcode on exit

As, in the previous experiment, the shellcode remains unchanged, but its execution is chained on the exit of the targeted binary.

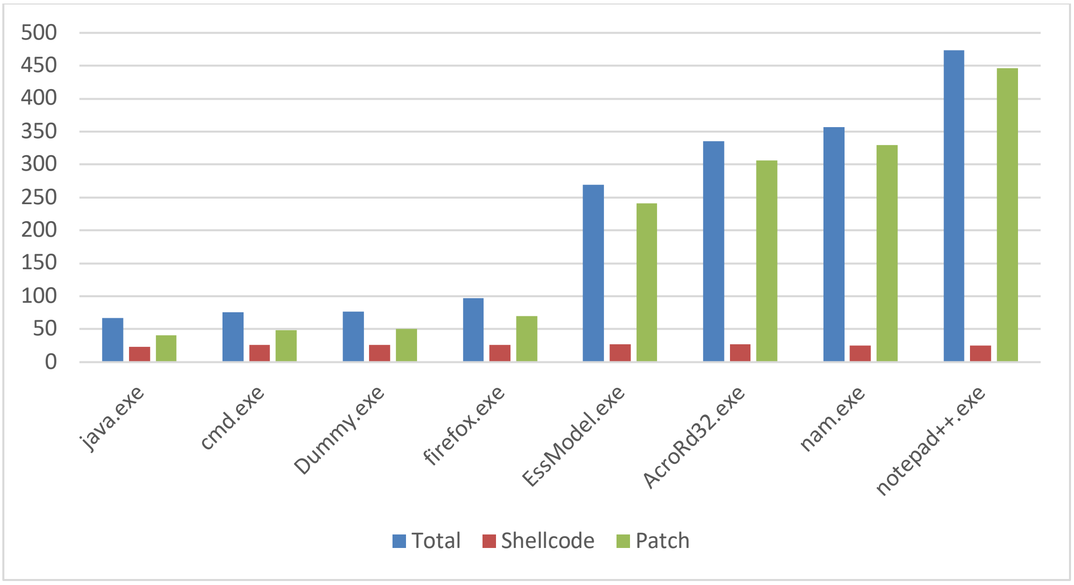


Figure 14. shellcode-revshell-exit mode execution times

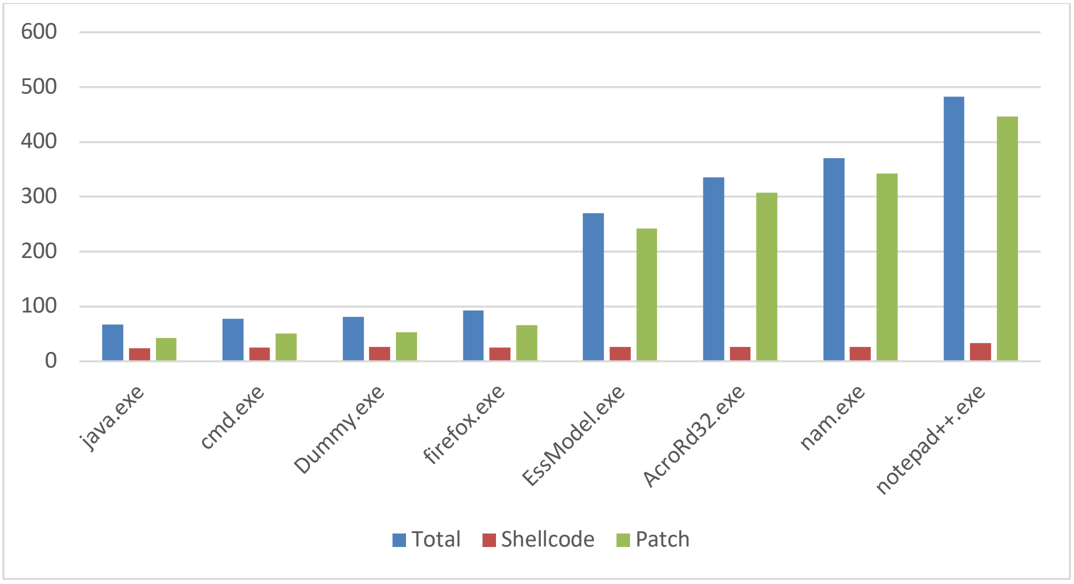


Figure 15. shellcode-revtcp-exit mode execution times

There no significant differences with the previous experiment as expected.

5.1.7 Overall

Considering the results of the tests, we can see two factors standing out and affecting the performance of the tool:

1. in the case where the shellcode is not ROP compiled, it is clear that the PE patching and the total time of execution depend totally on the size of the executable.

2. In the other case, where the shellcode gets compiled to ROP, we can see, that the time is affected not only by the size of the executable but also by the "compatibility" of the shellcode with the PE. Compatibility in this scope means

"how easy is for ROPInjector to match gadgets to the provided shellcode". This is why nam.exe, even if it is the second largest PE, the infection procedure is much shorter than other smaller binaries.

5.2 Similarity

In this section, we are examining the similarity between the original PE and the PE after patching. First of all, it is really difficult to define an exact measurement for similarity. The direction we followed, was to analyze the .TEXT section of the two PEs and compare them as strings. To compare them two algorithms for string similarity were considered:

- Cosine
- Jaro-Winkler.

The problem with the Cosine algorithm is that it does not take into account the ordering in the string.
The results can be found in the tables below:

revshell	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

Table 2. Cosine Similarity (revshell)

revshell	ROP Entry	ROP Exit	Shellcode Entry	Shellcode Exit
Dummy	0.9383196	0.9420777	0.9768985	1.0
firefox	0.9850798	0.9850998	0.9999644	0.9999932
java	0.8481479	0.8481479	1.0	1.0
notepad++	0.837363	0.837363	1.0	1.0

Table 3. Jaro Winkler Similarity (revshell)

revtcp	ROP Entry	ROP Exit	Shellcode Entry	Shellcode Exit
Dummy	0.9989467	0.9989456	0.9999999	1.0
firefox	0.9999808	0.9999807	0.9999921	0.9999917
java	1.0	1.0	1.0	1.0
notepad++	0.9992145	0.9992145	1.0000000	1.0000000

Table 4. Cosine Similarity (revtcp)

revtcp	ROP Entry	ROP Exit	Shellcode Entry	Shellcode Exit
Dummy	0.9419392	0.9381984	0.9768985	1.0
firefox	0.9862227	0.9862342	0.9999647	0.9999932
java	1.0	1.0	1.0	1.0
notepad++	0.8349906	0.8349906	1.0	1.0

Table 5. Jaro Winkler Similarity (revtcp)

From this analysis, it's clear that the .TEXT section of both the benign and infected PEs, are almost identical and it's fairly difficult to distinguish between them without knowing what to look for. This is explained be the fact that ROPInjector tries to apply minimal changes in the targeted PE and injects just the absolutely needed parts to ensure the execution of the shellcode.

5.3 Complexity

ROPInjector has a really big codebase and analyzing the whole program to calculate its complexity is not feasible. To bypass this limitation and get an estimation on the complexity of the code, first of all, we used lizard [3], a tool for static code analysis written in Python. Some indicative data can be found in the table below:

Function	NLOC	Cyclomatic Complexity	Number of Params	Length
classifyGadget	692	330	1	758
classifyInstruction	354	199	1	399
ropCompile	301	85	9	368
makeRel32Branch	114	28	3	130
patchPEInMemory	107	19	3	141
analyze	40	6	4	57

Table 6. Complexity stats for ROPInjector functions

Based on these results, it would be useful to describe the functionality of the two first functions. Both the shellcode instructions and the candidate gadgets found in the PE, are being classified into common categories based on their type. The equivalent procedures for instructions and gadgets are presented below:

5.3.1 classifyGadget

ROPInjector parses all the candidate gadgets, found in the PE, into an intermediate representation that makes it easier to perform one-to-one permutations. The purpose of this function is to arrange the candidate gadgets into categories, based on the type of instructions they could potentially encode. A special encoder function is assigned to each gadget. The encoder function is responsible to answer if the gadget could encode a given (already classified) instruction.

5.3.2 classifyInstruction

All ROP-able instructions of the provided shellcode are parsed into a higher-level representation. This is done to enable ROPInjector to better match each instruction with one of the candidate gadgets found in the attacked PE later on. Additionally, two of the most commonly used functions, are analyze and ropCompile. As described previously the former one is used to cast each instruction of the provided disassembly to ROPInjector's data structures and the latter one, to transform the provided shellcode to its ROP equivalent.

5.3.3 analyze

Although it is not one of the most complex functions, the fact that it is used so many times throughout the execution of the program, makes us believe that it has an effect to the overall complexity. This function's complexity is calculated to be at the worst-case $O(n^2)$ where n is the number of instructions is the provided disassembly.

5.3.4 ropCompile

We calculated the complexity of this function to be, at the worst case, $O(n^2)$ where n is the number of instructions in the provided shellcode.

5.4 Difficulty of ROP compilation

The difficulty of building the ROP equivalent can be affected by the size of the shellcode as well as the PE, as it can be seen in the bar charts presenting the times of execution for different PEs. On the other hand, a PE rich in instructions that can be used for chaining to the shellcode, could make the ROP compilation of the shellcode easier. Last but not least, we should comment on the difficulty introduced by the ROP permutations. Each instruction could potentially be replaced by one or more instructions. But in order to limit the difficulty of searching in the generated space by one-to-many permutations, it was decided to limit the search to one-to-one permutations. Keeping this in mind, in addition to the fact that ROPInjector must ensure that the execution of the original program will not be affected, the lack of free registers when encoding an instruction to its ROP equivalent, could affect the procedure.

6 Techniques against ROP attacks

Before listing techniques against ROP-attacks we should try to identify and understand the key characteristics that this kind of attacks hold:

- Since ROP attacks disturb the normal control flow of execution, they may increase the number of mispredicted branches by the processor branch predictor.
- ROP attacks may show different usage for ret and call instructions, as well as push and pop since they depend on chaining blocks of instructions that load data from the hijacked program stack to registers, and later return to the stack
- ROP attacks chain gadgets from arbitrary memory locations, so attacks may exhibit low memory locality when compared to clean execution.
- ROP attacks use the stack for chaining gadgets, and the gadgets are typically spread out across the memory of the program, thus they show lower reuse of the same memory blocks compared to clean executions.

Keeping these characteristics in mind, there is a variety of options for detecting ROP attacks. We will briefly describe some of them.

6.1 Stack Canaries

The "stack canaries" method places secret values in a control buffer and on the stack. The values are changed every time the protected program is started. In the epilogue of the functions and just before returning these values are tested against the control buffer. If the values do not match, then an unwanted change in the stack is detected and following protection actions can be taken.

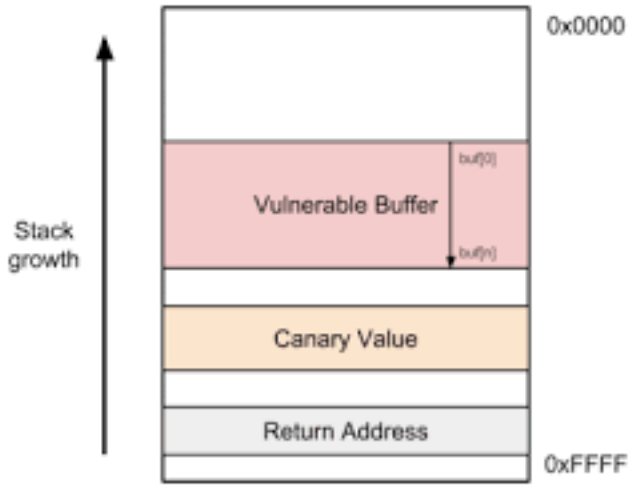


Figure 16. Stack canaries usage example

6.2 Shadow Stack

In this method, programs create a second stack that follows the program's execution flow. To accomplish that, in the prologue and apart from the primary stack, functions store their return address in the secondary "shadow" stack. Then and following the execution, in the epilogue, functions compare the two stored return addresses. In case the two compared addresses are different then the attack is detected. At this point developers can choose the action that will be taken against the threat. The options ranges from just terminating the program to alerting the system's administrators.

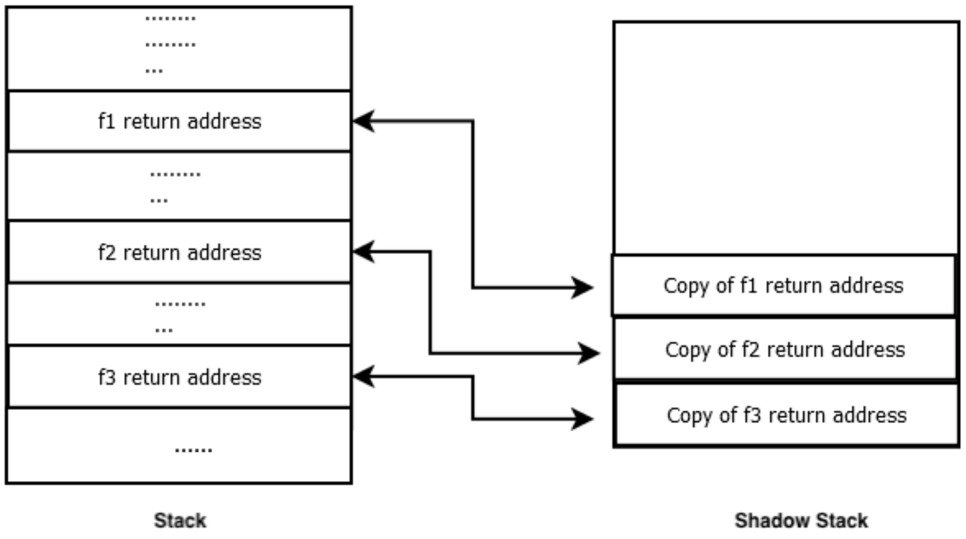
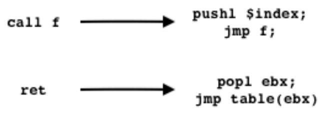


Figure 17. Shadow stack example

Attackers could bypass this measure of protection by overwriting the return address in the shadow stack too. But shadow stacks can be themselves hidden and protected against this kind of attacks. In this sense, shadow stacks can offer more protection compared to stack canaries, since the latter defense mechanism can be vulnerable to non-contiguous write attacks [5].

6.3 "Return-less" Kernels

By modifying compilers [6] to not use the "*call*" and "*ret*" instructions, the creation of ROP gadgets could be completely avoided. The idea is to create an immutable table which contains all the return addresses that are allowed in each program. This way, the creation of ROP gadgets would be prevented since all their building components will not exist. To achieve this all "*call*" and "*ret*" instructions would have to be replaced as shown below:



```
call f   ──────►   pushl $index;
                     jmp f;

ret      ──────►    popl ebx;
                   jmp table(ebx)
```

Figure 18. Return-less kernels

6.4 Last Branch Recording

Keeping in mind that ROP gadgets are chained using the *RET* instruction (in which typically they end), we could intercept calls to system functions and check how to the flow of the program was transferred to this point. The normal behavior would be using a call instruction. But in the case of a ROP attack, the last instruction would be a RET statement which is considered as an abnormal and alarming behavior. Modern processors have the ability to record jump addresses, using a mechanism called Last Branch Recording that could be used for detecting ROP attacks.

6.5 Pointer Authentication Codes

ARMv8.3-A architecture has a new feature for protection against ROP attacks. By using the spare bits in a pointer’s address space to cryptographically sign pointer addresses, each jump is checked and verified before it occurs. If the validation of the pointer’s address fails, then the jump is not performed.

7 Future Work

7.1 Types of binaries

As mentioned, currently, ROPInjector can only patch 32-bit Portable Executable files. As a next step, the implementation could be extended to be generic and architecture agnostic. To accomplish that, a first step would be to extract and define a higher-level API. Since the procedure mostly common for most architectures, this should be fairly easy to achieve.

7.2 Python Framework

The state of the PoC source code, makes it very difficult to implement generalizations or to add new features. A suggestion would be to reimplement the ROPInjector in a way that would allow the use of plugins according to each use case. A huge improvement towards this direction would be to use a higher-level language. Moreover, using an object-oriented language would allow the modeling of all the existing data structures in a way that is more user friendly. A Python implementation of ROPInjector, would hold all the great advantages mentioned before, adding also the benefit of the software's portability. We have to mention though, that the low-level memory manipulations are much easier as they are implemented in C.

8 Conclusions

In this thesis we tried to analyze the functionality of ROPInjector tool. We briefly presented Return-Oriented Programming as an alternative to polymorphism for AV evasion. Furthermore, we described the tool's functionality and focused on its key algorithms, performing a qualitative analysis for each of them.

Based on the previous analysis and by testing several Portable Executables, we attempted to identify the similarity between the benign and infected PEs, proving that in all cases, they are almost identical and almost to impossible to distinguish.

We measured the time of execution for multiple modes of operation trying to identify the parameters that affect the tool's performance. We identified the size of the targeted binary and the availability of ROP gadgets in its .TEXT section as being the characteristics standing out.

In the process of presenting techniques for protection against return-oriented attacks we tried to identify the main characteristics present in these attacks and proposed future work topics in order to make ROPInjector more usable to the research community.

Having these in mind, we can comment the following for the current implementation:

8.1.1 Patching portable executables

ROPInjector patched all tested PEs without a problem. As it can be shown in the bar charts presenting (execution time), the size of the PE might affect the execution time but there should be no other visible effect. ROPInjector is not designed to analyze other types of executables since all manipulations of the evaded executable are done with the assumption that the provided binary is in PE format.

8.1.2 Shellcodes

All analysis done in the provided shellcode is generic enough to handle shellcodes in text as well as in binary format. But we have to mention here the following assumptions taken by the implementer of the tool:

- There is a defined set of instructions that cannot be compiled to a ROP representation. Specifically, all instructions accessing ESP except for push immediate are not compiled.
- ROPInjector depends on the output of the ProView disassembler. There is the possibility that code ambiguity in the provided shellcode could lead to failures.

8.1.3 ROP Compilation

ROPInjector will encode one shellcode instruction using only one gadget instruction. This decision was made in order to eliminate the complexity introduced by calculating more than one permutation of a given shellcode instruction. Furthermore, in this proof-of-concept implementation, not all types of instructions are encoded into their return-oriented equivalents. Such instructions are branches (i.e. jumps, calls, loops, interrupts), privileged instructions and pops.

References

[1] G. Poulios, "Github," January 2019. [Online]. Available: https://github.com/gpoulios/ROPInjector.

[2] A. Honig and M. Sikorski, "Oreilly," January 2019. [Online]. Available: https://www.oreilly.com/library/view/practical-malware-analysis/9781593272906/ch02s05.html.

[3] Metasploit, January 2019. [Online]. Available: https://www.metasploit.com/.

[4] T. Yin, "Github," January 2019. [Online]. Available: https://github.com/terryyin/lizard.

[5] L. Szekeres, M. Payer, T. Wei and D. Song, January 2019. [Online]. Available: https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf.

[6] J. Li, Z. Wang, X. Jiang, M. Grace and S. Bahram, January 2019. [Online]. Available: https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/EUROSYS10.pdf.