University of Piraeus

Department of Digital Systems

Postgraduate Programme

«Techno-economic Management & Security of Digital Systems»

Master's Thesis

Advanced Antivirus Evasion Techniques

Poulios Giorgos

MTE/1127, gpoulios@unipi.gr

Under the supervision of:

Prof. Christos Xenakis, xenakis@unipi.gr

Dr. Christoforos Dadoyan, dadoyan@unipi.gr

Piraeus 2014-15

This thesis is dedicated to my parents, Katia, Thano, DK and VF
for their endless support, patience and belief in me

Many thanks to Dado for his guidance and fruitful discussions

Table of Contents

Table of Figures

Table of Tables

Abstract

In this thesis we examine the use of Return-Oriented Programming (ROP) combined with other practices for local (i.e. infected executables on disk) antivirus evasion. ROP is considered as a polymorphism alternative to *crypters* and *packers*. The software product of this work is a tool written in Win32 C which, given any piece of shellcode and any non-packed 32-bit Portable Executable (PE) file, it transforms the shellcode into its ROP equivalent and patches it into (i.e. infects) the PE file. After trying various combinations of evasion techniques, the results show that certain methods can evade nearly and completely all antivirus software employed in the online VirusTotal service. From a theoretical standpoint, the main outcome of this research is a) the algorithms for analysis and manipulation of assembly code on the x86 instruction set (up to and excluding the SSE), and b) the highlighting of common antivirus software weaknesses.

1. Motivation

The evolution of antivirus (AV) detection mechanisms starts with plain string signatures of common malware code found in exploits and viruses. Executable files on disk are traditionally scanned by AVs for matches against any of the signatures collected in their databases. The method is static and most of the time reactive, i.e. after a worm/virus outbursts, security analysts examine the malware code and create signatures for AV software to match against it. Malware authors proved this scheme trivial to bypass by injecting NOP (No-OPeration) instructions and code that cancels out itself or code that is never reached (a.k.a. dead code) in-between the malicious instructions. To tackle such constructs, signatures evolved to regular expressions (RegEx). With regex'es powerful capabilities, now AVs relied on "some" of the malicious instructions being there, even if irrelevant code was in-between. The next step by the attackers' side was to cypher the malicious code and decipher it online when taken control of execution flow. Such malware are known as polymorphic and/or oligomorphic. Even though the decryptor is still subject to signing and the technique requires a read-writable-executable (RWX) memory section, polymorphic code is still very popular and in many cases effective, especially when the encryption is run multiple times over the cipher. A special case of polymorphism is metamorphic and self-modifying code that mutates itself on each infection while maintaining the same functionality. Polymorphism can be tackled by a) static analysis, i.e. disassembly and Control Flow Graph (CFG) extraction, b) heuristics on the entropy and statistics of the machine code, and c) dynamic/behavioral analysis using emulation or virtualization, i.e. execution of the suspicious code into a sandbox environment in order to profile sequences of invocations to common system calls.

The main argument represented by this thesis is that Return-Oriented Programming (ROP) can be a strong polymorphism alternative (for the reasons outlined in Section 3) and that if combined with mild behavioral anti-profiling techniques it may render AV detection near infeasible. Besides the transformation of the code into a non-recognizable non-recurrent form though, several additional issues must be considered to achieve evasion such as the patching method. This includes, but is not limited to, the positioning of the virus in the carrier executable and the way of transferring control to the virus 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).

To prove the aforementioned concept, we choose to work on Win32 Portable Executables (PEs) and the x86 architecture by building a ROP compiler and patcher in Win32 C that converts any piece of given shellcode (hereafter , also referred to as *source (shell)code*) into its ROP equivalent and infects any given PE with it. Figure 1 summarizes the functioning of the proof-of-concept tool developed to infect and evaluate the evasion ratio of the proposed techniques.

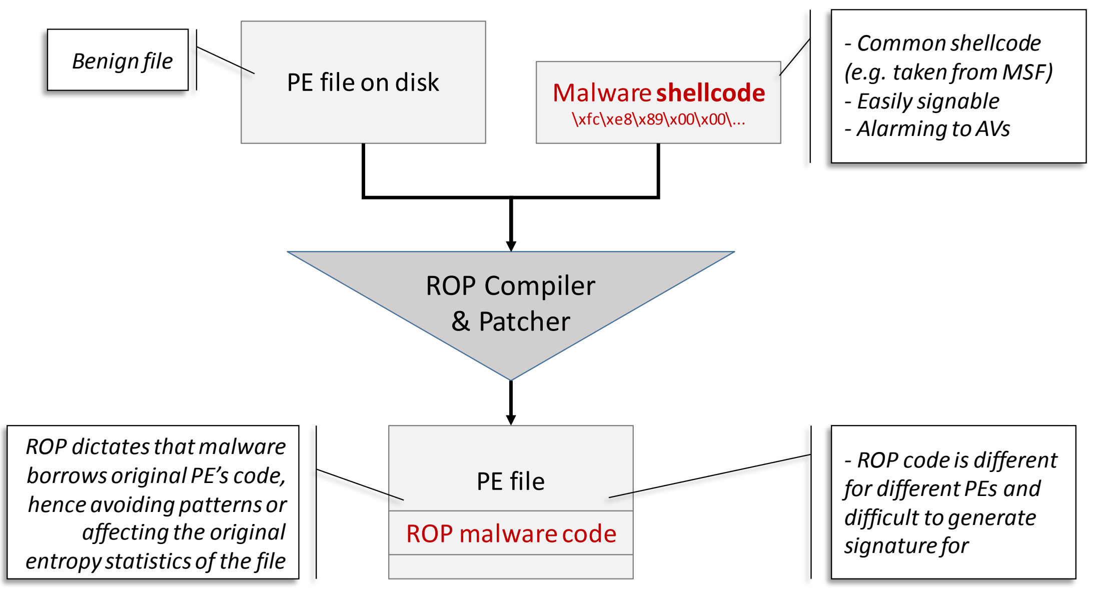

Benign file
PE file on disk
Malware shellcode
\xfc\xe8\x89\x00\x00\...
- Common shellcode (e.g. taken from MSF)
- Easily signable
- Alarming to AVs
ROP Compiler & Patcher
ROP dictates that malware borrows original PE's code, hence avoiding patterns or affecting the original entropy statistics of the file
PE file
ROP malware code
- ROP code is different for different PEs and difficult to generate signature for

Figure 1: Overview of the proof-of-concept ROP patcher

The rest of this paper is structured as follows: Section 2 is a brief positioning with respect to related work on the field of PE infection; Section 3 presents the approach to transforming normal x86 machine code to its return-oriented equivalent and the patching mechanisms; section 4 is a brief documentation of the developed software and section 5 presents evaluation of the results after infecting several well-known executables with some of the most common and popular shellcodes. Finally, section 6 concludes the thesis with discussion on the evolution of both AV detection and malware evasion with respect to ROP.

2. Related Work

To the author’s best knowledge this is the first work that infects PEs with ROP-encoded payload. Nevertheless, in this section we examine two tools having the same purpose with our ROP patcher, that is, to infect PE files with common (possibly encrypted) shellcode in a way that bypasses AV software.

The first, 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, as detailed later on paragraph 3.1, the proposed approach introduces variability too, due to the transformation to ROP (which is dependent on the PE file).

PEinject [2] 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 nests or section padding) and does not encrypt or modify the shellcode 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.

The evasion ratios of both methods are compared with the proposed approach in Section 5.

3. Approach

3.1 Return-Oriented Programming as a polymorphism alternative

ROP gained increased attention during the late 2000's [3] as an advanced stack smashing attack that could bypass Data Execution Prevention (DEP) mechanisms. It 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 (commonly `ret`, 0xC3, but not only). 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.

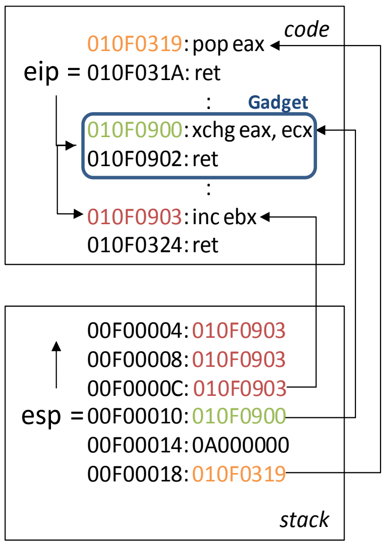


Figure 2: Sample return-oriented program performing
`mov eax 10; xchg eax, ecx; add ebx, 3`

The first and most important benefit of using ROP for AV evasion is that such borrowed code (that of gadgets) is always benign and tested against false positives. 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 <VA`$_i$`>` operations, and c) can be randomized or encrypted in many and trivial ways. (b) in particular 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.

Given these features, ROP enables polymorphism **without requiring a writeable code section in memory** (which is very rare in benign PEs unless they are packed, as well as a typical heuristic for detection). Encryption/decryption can be applied on the gadget chain in memory (i.e. in the stack and not in the code section) and/or different gadgets can be randomly chosen for the same operation hence altering the malware's footprint.

3.2 Other techniques employed for antivirus evasion

Besides transforming the malicious code into its return-oriented equivalent for hiding it, a few other techniques are necessary or complementary to achieve lesser detection ratios.

3.2.1 PE patching and passing control to the malware

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 malware 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 malware code in the 0xCC padding commonly left by the linker in-between code segments (typically OBJ files) in the .text section of PEs. However, a) there may not always be sufficient space in those CC nests, and b) we will be using them for our ROP purposes later on.

For these reasons we choose to append the malware 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 those replaced instructions followed by a jump back to the original execution flow. Figure 3 (a) depicts this scenario. Directly pointing the address of entry point to the malware in this case is avoided since many AVs' heuristics are alarmed by the fact that it points towards the end of .text. An alternative to giving control to the malicious code at program entry, is to hook any calls to `ExitProcess`, `exit` or other similar function as depicted in Figure 3 (b). This technique in particular, as shown also later by the results, bypasses behavioral profiling by AVs that employ emulation or sandboxing. To our best knowledge, that is either because AVs emulate only a small portion of the executable's entry code due to scanning time constraints, or because of lack of (universal) techniques for triggering a graceful exit (most programs do not handle SIGINT and SIGTERM signals).

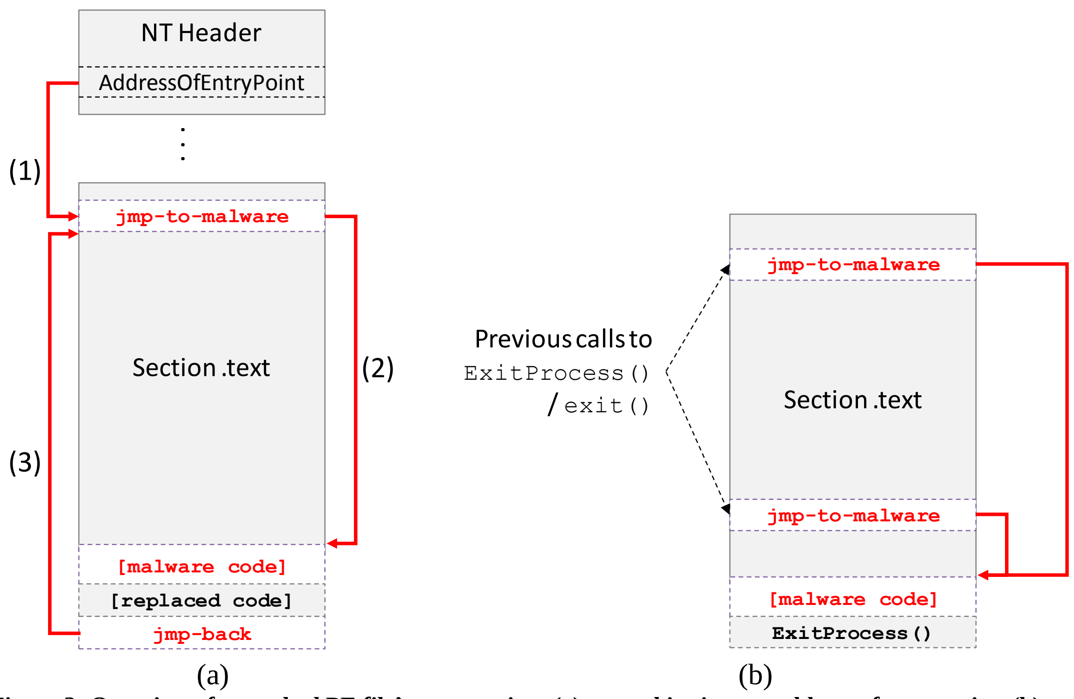


Figure 3: Overview of a patched PE file's .text section: (a) control is given at address of entry point, (b) control is given right before the program exits.

3.2.2 Delaying execution

Delaying execution of the malicious code by "sleeping" a few seconds or minutes was attempted as a means to bypassing behavioral profiling -especially when control is given during the program entry- with unsuccessful results. Our conjecture is that during emulation, calls to Win32 `Sleep()` may be intercepted and avoided in order to speed up the scanning process.

3.2.3 Replacing `getPC` constructs

Shellcode is usually built for remote exploitation during which the attacker does not know the value of the program counter (EIP register), which is often necessary for memory addressing and callbacks. Exploit authors have been long using for this purpose some special (`getPC`) constructs to get the value of EIP. Among them the most common (and notably uncommon in regular machine code) is the relative `call` to a `pop` instruction (the `call` instruction pushes the value of EIP onto the stack and the `pop` retrieves it). However, when statically patching a PE file, EIP can be pre-calculated by knowing the (preferred) image base and adding a reference to the relocations table of the PE. Hence, to further decrease the suspicion ratio we replace such calls to `getPC` with:

```
 1: push <VA-of-8>
 6: jmp-to-pop
 8: ...
18: pop r32
```

and add the VA of `<VA-of-8>` (2 in this case) to relocations so that it gets repaired by the windows loader in case the PE cannot be loaded in the preferred image base.

3.2.4 Hiding the certificate

An issue that arises when patching signed executables is that their checksum/hash, and thus their certificate, gets invalidated. This is obviously very alarming and would prevent us from testing our methods on popular executables of every-day use. Surprisingly though, hiding the certificate by erasing its pointer in the security data directory of `NT_HEADER` does not trigger any alarms. Of course, regardless of the certificate, the checksum of the PE file is re-calculated and patched accordingly.

3.3 Reverse analysis of x86 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 functioning. 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 an unsafe instruction is any branch, or any using indirect addressing because it risks raising an access violation error).

Other useful information collected during the reverse analysis phase include the offsets (within the opcode) to the MOD/REG/RM byte, Scaled Index Byte (SIB), displacement, immediate constant, primary opcode and prefixes. Such constructs are preprocessed to infer the bitness of the operand(s), whether it uses indirect addressing, what registers it reads and writes, and whether it contains a direct Virtual Address (VA) reference. VA references are needed when modifying the original PE's code and particularly useful when pushing the VAs of gadgets onto the stack because they must be repaired in the relocations table.

Section 4 presents in detail the data structures used to hold this information as well as the algorithm used for inferring the ranges of free general purpose registers in the shellcode. The semantics considered throughout this paper (as well as the PoC implementation) regarding the state of general purpose registers are listed in Table 1.

Table 1: Semantics on general purpose register access

Notation/Predicate	Meaning
READS(I, X)	Instruction *I* reads the value of register *X*
WRITES(I, X)	Instruction *I* writes/modifies the value of register *X*
ACCESSES(I, X)	*READS(I, X)* ∨ *WRITES(I, X)*
SETS(I, X)	*WRITES(I, X)* ∧ ¬*READS(I, X)*: Instruction *I* sets the value of register *X* without considering its previous value
FREE(I, X)	Register *X* is free during execution of instruction *I*; any modification of *X* during *I* will not alter execution flow or results in any way

Based on this formalism, a register *X* is considered *linearly free* in a range of instructions $[I_a, I_b]$ iff:

- $\neg\exists\ I \in [I_a, I_b) : READS(I, X)$, i.e. no instruction reads *X* up until I_b, and
- $SETS(I_b, X)$

The lower bound I_a is chosen to be the last instruction prior to I_b that reads *X*. Note also how upper bound I_b might not be the first instruction that sets *X*.

The proposed method for determining whether a register is *totally free* (as in *FREE(I, X)*), requires inferring such linear ranges and then recursively following relative branches in the code to potentially reduce the linear ranges according to register access in the branch destination(s) (the algorithm is documented in paragraph 4.2.2).

3.3.1 Preprocessing of relative branches

It is often required during the transformation of the source shellcode into ROP (and the prerequisite steps) to insert additional instructions or replace them with more/longer ones resulting in increased bytes in-between relative branches. This introduces the risk of overflow when repairing the offsets of relative branching instructions which are typically encoded in the least number of bytes possible (often 8-bit offsets). Testing and repairing such relative offsets on a per modification basis might result (unlikely but possible) in a cascading effect where one repair to avoid overflow (e.g. replacing a `jmp rel8` /2 bytes, with a `jmp rel16` /4 bytes) causes another overflow. To avoid this situation we convert all relative branches to their 32-bit equivalents prior to any modification of the shellcode. If the size of the resulting patch is an issue, the process can be easily inverted in the end.

3.3.2 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:

i) 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;
ii) They often read many general purpose registers at once, thus reserving them while as mentioned earlier, the more the free registers the better;
iii) 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, `[1] mov eax, [ebx+ecx*2]` would be replaced by:

```
[1']     sal ecx, 1
[2']     add ecx, ebx
[3']     mov eax, [ecx]
```

which requires that *FREE(1, ecx)* holds, otherwise an alternative is to replace with:

```
[1']     mov eax, ecx
[2']     sal eax, 1
[3']     add eax, ebx
[4']     mov eax, [eax]
```

which requires that *SETS(1, eax)*. If none of the above holds then another temporary register that is free may be used, and if there is no such register we abort the conversion.

Noteworthy is how unrolling unlocks register access from one atomic instruction to many. For instance, in the latter example, ecx is freed at [1'] and ebx at [3']. If eax where free at the preceding, say, 10 instructions, [1'] through [3'] could be moved 10 instructions behind, thus resulting in one more free register (ecx and ebx, minus eax which will not be free then) in that preceding code segment.

3.4 Compilation to ROP

Compilation of the source shellcode to its return-oriented equivalent involves searching for, injecting, and chaining gadgets in the benign PE file as well as permuting instructions to enhance encoding results. In addition, and in order to assist permutations and encoding, both gadgets and instructions are parsed into a higher level intermediate representation (IR) abstracting out the peculiarities of the machine code and the architecture.

3.4.1 Finding gadgets

Candidate gadgets in the executable sections of the given PE file must end in one of the following chaining instructions (hereafter also referred to as *gadget endings*, or simply *endings*):

- `ret`
- `retn`
- `pop regX; jmp regX`
- `jmp regX`

Exceptionally for the latter, the gadget in question must be then paired with a *loader gadget* of the form:

```
[1]      pop regX
[2]      <any of the first 3 endings>
```

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 `push` instruction are filtered out. The last step of gadget finding is to filter out exact duplicates (byte-by-byte comparison) for performance reasons (although they could serve randomization purposes later on).

3.4.2 Parsing gadgets into IR

The gadgets found in the aforementioned process are first analyzed instruction-by-instruction as presented in 3.3 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. a `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 parsed into an IR consisting of a type, and 3 operands with different meaning depending on the type. Table 2 lists the types of gadgets and instructions defined by this IR, their semantics and the eligible instructions that are classified into each type. If a multi-instruction gadget contains more than one representable instructions, only the first is considered. However, the ones following have also been considered in other gadgets with the same ending because of the backwards gadget finding process described in the previous paragraph.

Table 2: Intermediate representation of gadgets and instructions

Type	Semantics	op1	op2	op3	Eligible instructions
LOADS	lods	eax	esi	bitness	lods (e)ax/al, m8/16/32
LOAD_REG	pop regA	regA	-	-	pop r32
LOAD_RM	pop [regA]	regA	-	-	pop m32
ADD_IMM	regA += imm	regA	imm	op1 bitness	add r8/16/32, imm8/16/32 add (e)ax/al, imm8/16/32 xor r8/16/32, 0 cmp r8/16/32, 0 inc r8/16/32 test r_a32, r_b32 (with r_a == r_b) test r8/16/32, 0xFF/FFFF/FFFFFFFF test (e)ax/al, 0xFF/FFFF/FFFFFFFF or r_a32, r_b32 (with r_a == r_b) and r_a32, r_b32 (with r_a == r_b)
SUB_IMM	regA -= imm	regA	imm	op1 bitness	sub r8/16/32, imm8/16/32 sub eax/al, imm8/16/32 dec r8/16/32
MUL_IMM	regA= regB*imm	regA	regB	imm	imul r32, r32, imm8/16/32 sal/shl r32, imm8 sal/shl r32, 1
DIV_IMM	regA /= imm (integer div.)	regA	imm	op1 bitness	shr/sar r32, imm8 shr/sar r32, 1
MOV_REG_IMM	mov regA, imm	regA	imm	op1 bitness	mov r8/16/32, imm8/16/32 imul r16/32, r16/32, 0 xor r_a8/16/32, r_a8/16/32 and r8/16/32, 0 and (e)ax/al, 0 or r8/16/32, 0xFF/FFFF/FFFFFFFF or (e)ax/al, 0xFF/FFFF/FFFFFFFF sub r_a8/16/32, r_a8/16/32
MOV_REG_REG	mov regA, regB	regA	regB	bitness	mov r8/16/32, r8/16/32 imul r16/32, r16/32, 1
MOV_REG_RM	mov regA, [regB]	regA	regB	bitness	mov r8/16/32, m8/16/32
MOV_RM_REG	mov [regA], regB	regA	regB	bitness	mov m8/16/32, r8/16/32
MOV_RM_IMM	mov [regA], imm	regA	imm	bitness	mov m8/16/32, imm8/16/32 and m8/16/32, 0 or m8/16/32, 0xFF/FFFF/FFFFFFFF
ADD_REG	regA += regB+x	regA	regB	x	[ADD/SUB_IMM]; add r32, r32
SUB_REG	regA -= regB+x	regA	regB	x	[ADD/SUB_IMM]; sub r32, r32
MUL_REG	regA *= regB	regA	regB	-	imul r32, r32 imul edx, eax, r32
DIV_REG	eax /= regC (integer div.)	eax	eax	regC	idiv edx, eax, r32
XCHG_REG_REG	xchg regA, regB	regA	regB	bitness	xchg r8/16/32, r8/16/32 xchg r8/16/32, (e)ax/al (with r != eax)
XCHG_REG_RM	xchg regA, regB	regA	regB	bitness	xchg m8/16/32, r8/16/32
GPUSH_IMM	push imm32	imm	-	4	*for gadget*: mov, [esp+x], imm32 (with x being 4+stackAdvance after gadget has returned) *for instruction*: push imm32
GPUSH_REG	push regA	regA	-	4	*for gadget*: mov, [esp+x], r32 (with x being 4+stackAdvance after gadget has returned) *for instruction*: push r32
UNDEFINED	[all others]	-	-	-	[all others]

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 (typically "pushes" of gadget VAs and immediate constants) if requested to. Given this IR, for instance, an encoder for an XCHG_REG_REG gadget might respond positively to a request to encode an XCHG_REG_REG instruction even if their operands are flipped. More details on such permutations are given in paragraph 3.4.4. The special case of an UNDEFINED gadget encoder requires that the given instruction to encode is an exact match at the byte level with the gadget.

3.4.3 Injecting gadgets

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 nests 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 nest, the gadget is injected in-between the preceding code and the epilogue. For instance, if the missing gadget were `mov ecx, eax` and the following code were found in the PE:

```
      <other instructions>
      mov esp, ebp
      pop ebp
      ret(n)
      CCCCCCCCCCCCCCCCCC
```

then the injection would result in:

```
      <other instructions>
      jmp epilogue          ;  for normal flow to avoid gadget instruction
      mov ecx, eax          ; the injected gadget instruction
      jmp return            ;  for gadget flow to avoid the standard epilogue
epilogue:
      mov esp, ebp
      pop ebp
return:
      ret(n)
      CCCCCCCC
```

In the case that no epilogue is found at the boundary with the 0xCC nest, 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:

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

Following gadget insertions will then reuse this pseudo-epilogue as stated above, by injecting before the standard epilogue, thus making it look more like a real function.

3.4.4 Source code permutations

We distinguish between two kinds of permutations: one-to-one, and one-to-many. The former is about one instruction being replaced by another, while the latter is about one instruction being replaced by many. The search space generated by the later quickly scales exponentially as one may seek for *n*-th order permutations recursively and its investigation is beyond the scope of this thesis.

Predefined, one-to-one permutations are achieved through the IR and encoder functions. Besides the ones deriving from the listing in Table 2, the 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.

3.4.5 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 (also chosen as an implementation option). 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 `pop`s, `retn`s) and gadgets with loader-gadgets (see also paragraph 3.4.1). 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. Of course, this list is produced in the direction of the source shellcode, while the chain building operations must be performed backwards in order to be executed forwards.

In order to counterbalance potential stack pointer modifications (mainly `pop`s, i.e. additions) by gadgets, the GADGET data structure (refer to Table 5 and Table 6 for details) contains information describing the number of bytes the stack pointer is added **before** (the case in `pop-pop-ret` kind of endings) and **after** the return (the case in `retn` kind of endings) of the gadget. Using this information, each encoder outputs ADVANCE operations in-between others accordingly. For instance, if a gadget were:

```
[1]       0x00040010: mov ecx, eax ; the useful instruction
[2]       ..........: pop ebp      ; redundant pop
[3]       ..........: retn 10      ; another 10 bytes added to esp after the
                                                                    return
```

then the encoding output would be:

```
PUSH_VA    0x00040010      ; VA of mov ecx, eax
ADVANCE    4               ; for pop ebp
CHAIN                      ; placeholder for VA of next gadget
ADVANCE    10              ; for retn 10
```

Its (backwards) interpretation by the compiler would result in the following chain building instructions:

```
[1]       push <VA of gadget following next>
[2]       sub esp, 10                ; ADVANCE 10
[3]       push <VA of next gadget>   ; CHAIN
[4]       sub esp, 4                 ; ADVANCE 4
[5]       push 0x00040010            ; PUSH_VA  0x00040010
```

In the case of multiple calls to the same gadget (e.g. as in using `inc eax` to achieve `add eax, X`) the compiler wraps the call with a conditional jump loop using a free register.

Furthermore, not all types of instructions can be easily encoded into ROP. In our proof-of-concept we do not encode 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
build_chain:
   [3]       push <VA of gadget N>
   [4]       ....
   [5]       push <VA of gadget 1>
past_the_chain:
   [6]       <other instructions / chains>
```

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

4. Software Documentation

This section is intended to provide an abstract documentation in terms of data structures and algorithms used in the implemented proof-of-concept rather than a reference documentation for developers.

4.1 Main data structures

Table 3 lists the fields of the INSTRUCTION data structure (also used as a linked list item). A list of such instructions is populated and returned as a result of the disassembly and reverse analysis of any machine code (either the shellcode's or gadget instructions').

Table 3: INSTRUCTION data structure

`typedef struct INSTRUCTION`	
`BYTE data[16]`	the complete instruction opcode lies in here
`WORD offsets`	to-primary-opcode | to-MOD/REG/RM | to-displacement | to-immediate 4 bits each
`BYTE regReads`	1-bit flag for each general purpose register this instruction reads
`BYTE regWrites`	1-bit flag for each general purpose register this instruction writes
`BYTE freeRegs`	1-bit flag for each free general purpose register during this instruction
`DWORD index`	the index of this instruction in the containing code segment
`INSTRUCTION *jmp`	pointer to the instruction it jmps/calls/loops to (if any)
`INSTRUCTION *next`	the next instruction in the containing code segment
`DWORD *directVA`	pointer to a VA in the INSTRUCTION.data array this instruction refers to (if any)
`DWORD flags`	e.g. HAS_SIB, IS_BRANCH/_{JMP, COND, REL, INT}, CONTAINS_VA, HAS_2B_OPCODE, HAS_OPCODE_EXT, HAS_8BIT_OPERAND
`BYTE totalSize`	total size in bytes of this instruction

Table 4 lists the fields of the more abstract IR for instructions and gadgets. Operands might represent registers, immediate constants or bitness of instruction/other-operands depending on the type field. For more details on the semantics and mapping to IR the reader is referred to section 3.4.2 Table 2.

Table 4: GINSTRUCTION data structure

`typedef struct GINSTRUCTION`	
`INSTR_TYPE type`	the type of this instruction (see Table 2 for possible types)
`long operand1`	the first operand in this instruction (also denoted as op1)
`long operand2`	the second operand in this instruction (also denoted as op2)
`long operand3`	the third operand in this instruction (also denoted as op3)
`INSTRUCTION *i`	pointer to the original instruction this IR originated from

Table 5 lists the fields of the gadget ending (GADGET_END) data structure as these are returned from the gadget finding process. Endings might contain the standard function epilogue before the return instruction as long as a corresponding gadget loader is found. `stackAdvBef` is used in case the return instruction is preceded by `pop`s while `stackAdvAft` is used in case the return instruction is a `retn`. The `reg` field is only used in case the ending uses a `jmp reg` instruction to return.

Table 5: GADGET_END data structure

`typedef struct GADGET_END`	
`DWORD va`	the VA of this ending in the host PE file
`DWORD numIns`	the number of instructions in this ending
`BYTE size`	the total size in bytes of this ending
`BYTE regWrites`	1-bit flag for each general purpose register this ending writes (join of all instructions in the ending)
`WORD stackAdvBef`	the number of bytes the stack pointer is advanced (added) before the actual return

`WORD stackAdvAft`	the number of bytes the stack pointer is advanced (added) after the actual return
`GEND_TYPE type`	the type of this ending (one of { RET, RETN, JMP })
`BYTE reg`	the register used to jump in case this is an ending of type JMP

Table 6 lists the fields of the GADGET data structure as these are returned from the gadget parsing process. The encoder function is assigned on a per-type basis, while it accepts as one of its arguments the gadget itself so that it can consider the value of operands (`gi`), modifications to registers, stack pointer adjustments and loader gadgets (if any).

Table 6: GADGET data structure

`typedef struct GADGET`	
`DWORD va`	the VA of this gadget in the host PE file
`GINSTRUCTION gi`	the IR of this gadget
`ENCODER encode`	pointer to the encoder function of this gadget
`DWORD flags`	e.g. MODIF_EFLAGS, INJECTED, SETS_DF, CLEARS_DF, LAST_IN_CHAIN
`BYTE regWrites`	1-bit flag for each general purpose register this gadget writes (join of all instructions and ending)
`WORD stackAdvance`	the number of bytes the stack pointer is advanced (added) after the useful instruction
`INSTRUCTION *i`	pointer to the list of instruction of this gadget (including ending)
`DWORD numIns`	the number of instructions in this gadget (including ending)
`GADGET_END *end`	pointer to the ending of this gadget
`GADGET *loader`	pointer to the loader gadget (applicable if ending returns using a `jmp r32` with no loader: `end.type == JMP && end.numIns == 1`)

Noteworthy at this point is the ENCODER function prototype. In C language it reads:

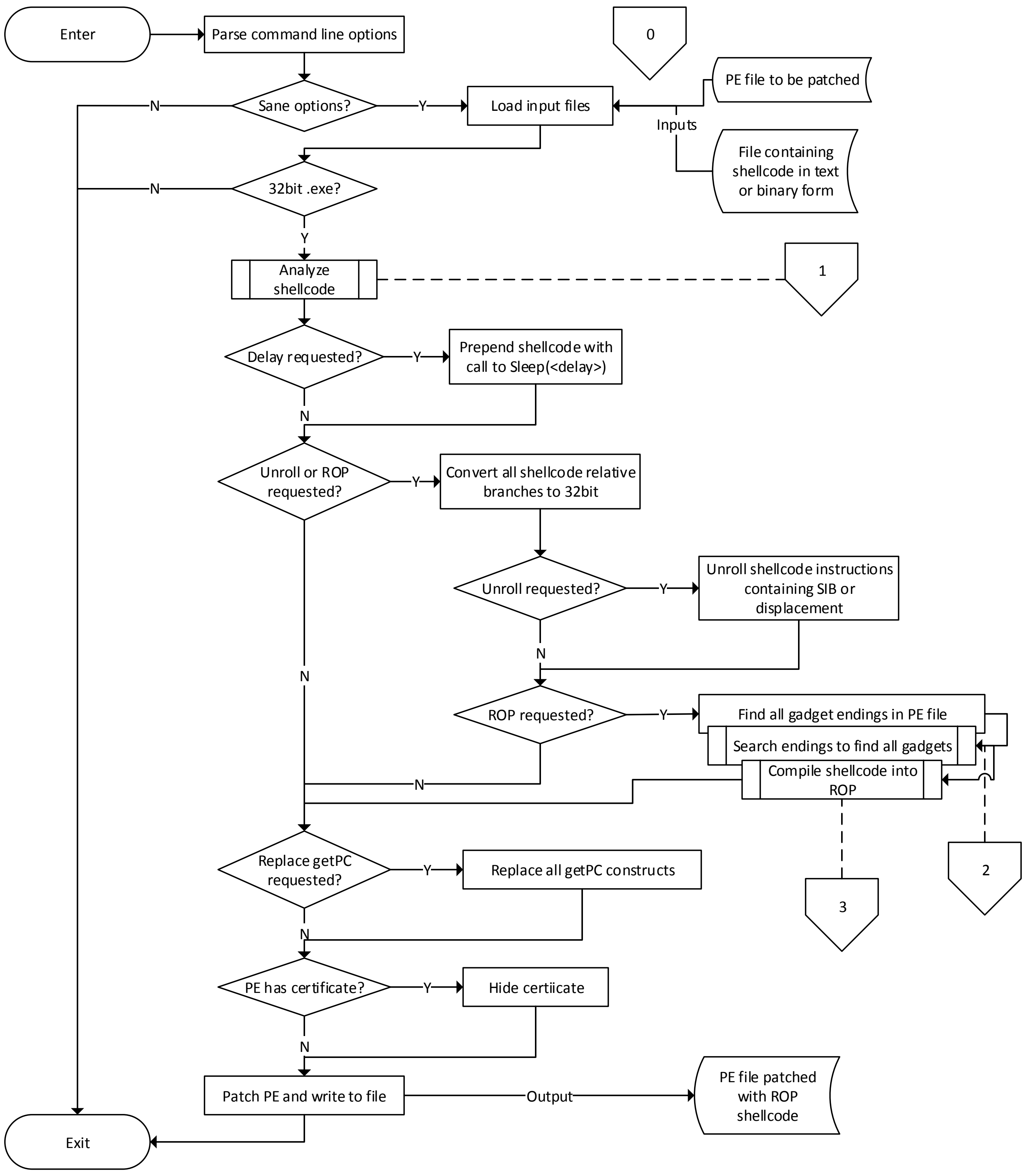
```
STACK_OPER*(*ENCODER)(const GADGET*const g, const GINSTRUCTION i, int *res);
```

That is, all encoders return a list of stack operations (as described in paragraph 3.4.5) and accept as input a gadget and an instruction, both classified into IR. The encoding result is stored by-reference in the 3rd argument. In case of failure, null is returned.

4.2 Algorithms and flowcharts

4.2.1 Main program

Figure 4 (flowchart 0) depicts the main process of the PoC ROP compiler implemented. It includes the parsing of the command line as well as decisions (based on the user's request) to delay execution of the shellcode (via `Sleep()`), to convert relative branches to 32-bit equivalents, to unroll, compile to ROP, replace `getPC` constructs, hide certificate and patch into the host PE. The option to omit compilation to ROP is used to evaluate the effect of ROP on AV evasion.



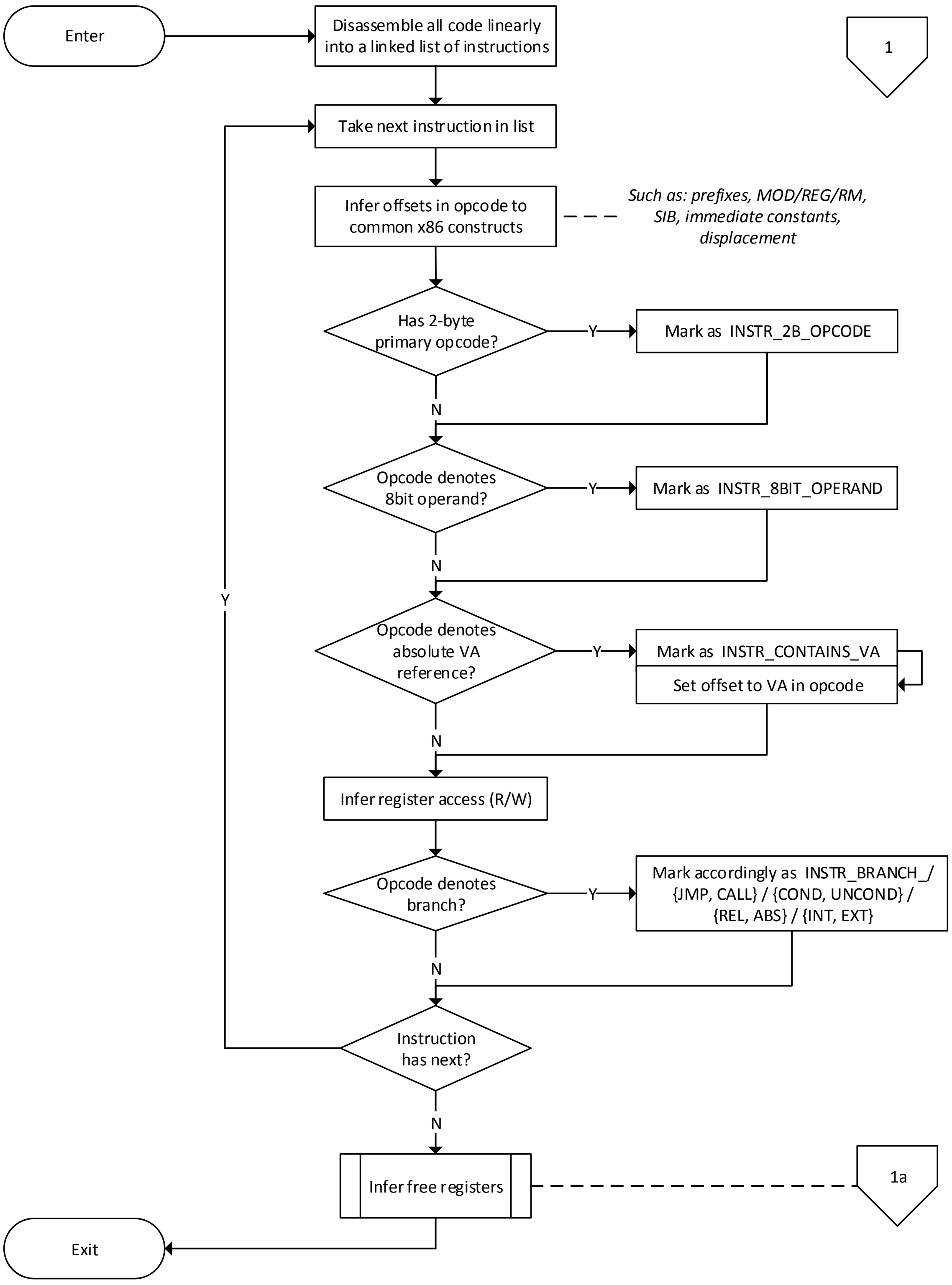
Enter
Parse command line options
0
PE file to be patched
Sane options?
N
Y
Load input files
Inputs
File containing shellcode in text or binary form
32bit .exe?
N
Y
Analyze shellcode
1
Delay requested?
Y
Prepend shellcode with call to Sleep(<delay>)
N
Unroll or ROP requested?
Y
Convert all shellcode relative branches to 32bit
Unroll requested?
Y
Unroll shellcode instructions containing SIB or displacement
N
N
ROP requested?
Y
Find all gadget endings in PE file
Search endings to find all gadgets
Compile shellcode into ROP
N
2
3
Replace getPC requested?
Y
Replace all getPC constructs
N
PE has certificate?
Y
Hide certiicate
N
Patch PE and write to file
Output
PE file patched with ROP shellcode
Exit

Figure 4: Flowchart 0, the main process of the ROP compiler tool

4.2.2 Reverse analysis of machine code

Figure 5 (flowchart 1) depicts the subprocess of x86 machine code analysis. For disassembly, the ProView disassembler [5] was used, while the ref.x86asm.net database was heavily consulted as x86 assembly reference. The marking of instructions that contain VA references is necessary for adding-to/repairing the relocations table. The inference of register access as stated at this stage is per instruction and is the subprocess that populates the

`INSTRUCTION.regReads/Writes` fields. The return value is a linked list of `INSTRUCTION`s.



Enter
Disassemble all code linearly into a linked list of instructions
1
Take next instruction in list
Infer offsets in opcode to common x86 constructs
Such as: prefixes, MOD/REG/RM, SIB, immediate constants, displacement
Has 2-byte primary opcode?
Y
Mark as INSTR_2B_OPCODE
N
Opcode denotes 8bit operand?
Y
Mark as INSTR_8BIT_OPERAND
N
Y
Opcode denotes absolute VA reference?
Y
Mark as INSTR_CONTAINS_VA
Set offset to VA in opcode
N
Infer register access (R/W)
Opcode denotes branch?
Y
Mark accordingly as INSTR_BRANCH_/ {JMP, CALL} / {COND, UNCOND} / {REL, ABS} / {INT, EXT}
N
Instruction has next?
N
Infer free registers
1a
Exit

Figure 5: Flowchart 1, disassembly and reverse analysis of x86 machine code

Figure 6 (flowchart 1a) depicts the wrapper function of calculating the free register ranges in the code, which in turn are used to set the `INSTRUCTION.freeRegs` field of all instructions.

Subprocess 1a1 calculates the linearly free registers while 1a2 repairs those ranges with respect to internal branches (relative jumps, loops and calls).

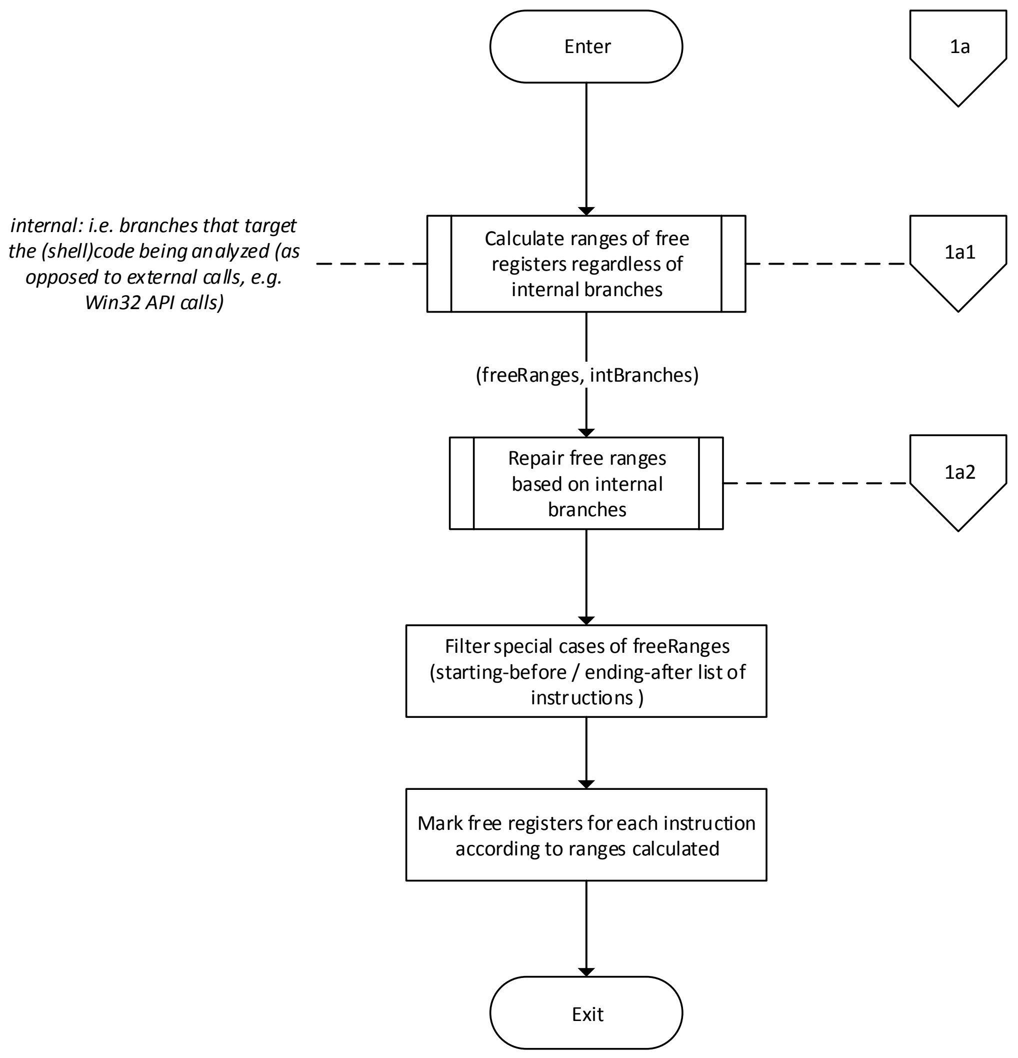

Enter
1a
internal: i.e. branches that target the (shell)code being analyzed (as opposed to external calls, e.g. Win32 API calls)
Calculate ranges of free registers regardless of internal branches
1a1
(freeRanges, intBranches)
Repair free ranges based on internal branches
1a2
Filter special cases of freeRanges (starting-before / ending-after list of instructions)
Mark free registers for each instruction according to ranges calculated
Exit

Figure 6: Flowchart 1a, calculation of free register ranges

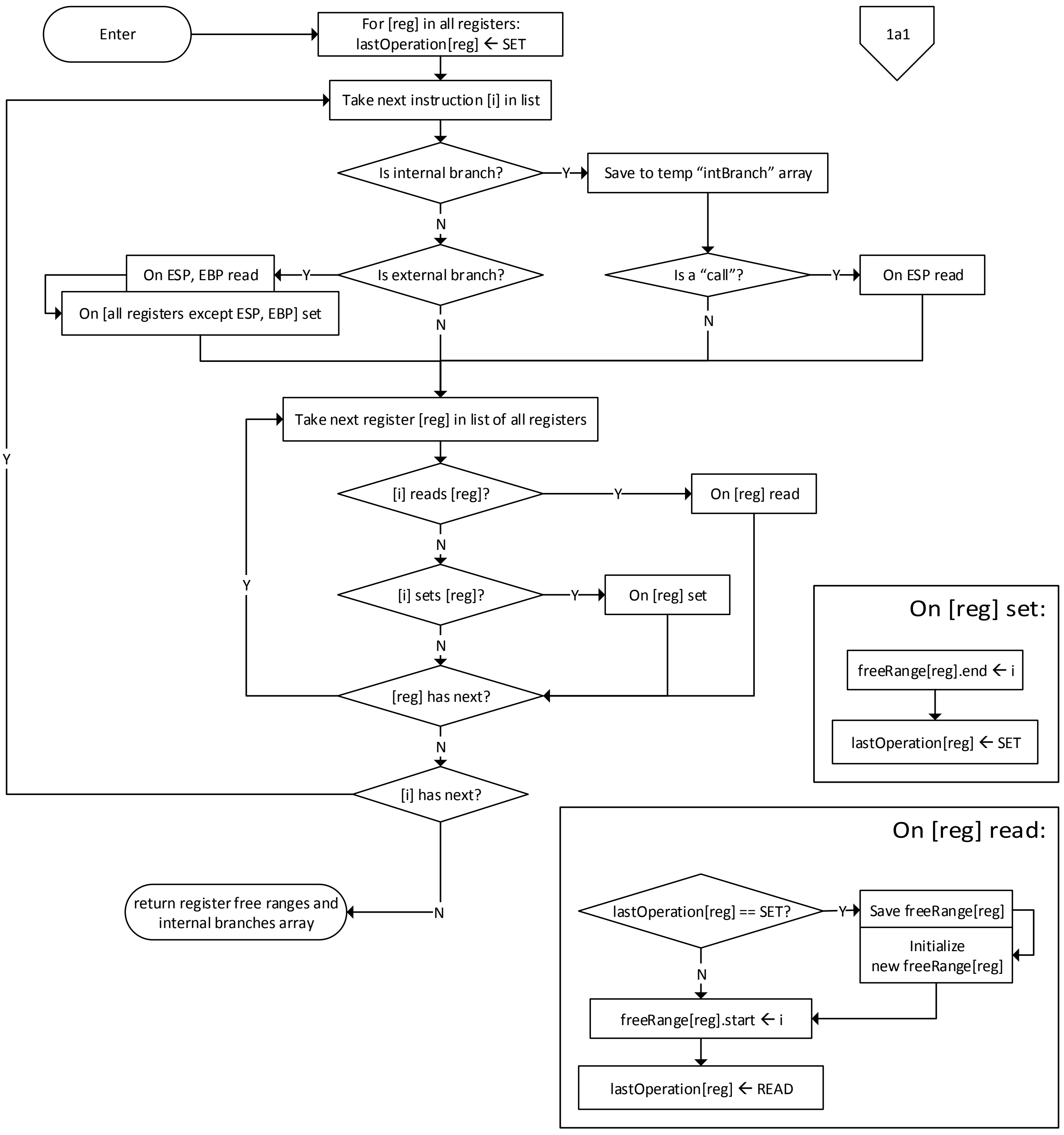

Enter
For [reg] in all registers: lastOperation[reg] ← SET
1a1
Take next instruction [i] in list
Is internal branch?
Y
Save to temp "intBranch" array
N
Is external branch?
Y
On ESP, EBP read
On [all registers except ESP, EBP] set
N
Is a "call"?
Y
On ESP read
N
Take next register [reg] in list of all registers
[i] reads [reg]?
Y
On [reg] read
N
[i] sets [reg]?
Y
On [reg] set
N
[reg] has next?
Y
N
[i] has next?
Y
N
return register free ranges and internal branches array
On [reg] set:
freeRange[reg].end ← i
lastOperation[reg] ← SET
On [reg] read:
lastOperation[reg] == SET?
Y
Save freeRange[reg]
Initialize new freeRange[reg]
N
freeRange[reg].start ← i
lastOperation[reg] ← READ

Figure 7: Flowchart 1a1, calculation of linearly free registers

Given the free register ranges -an array (freeRanges) of linked lists one per register- and an array of internal branches (intBranches), both calculated by subprocess 1a1, the next subprocess (1a2), repairs (i.e. shrinks) those ranges according to relative branching in the code.

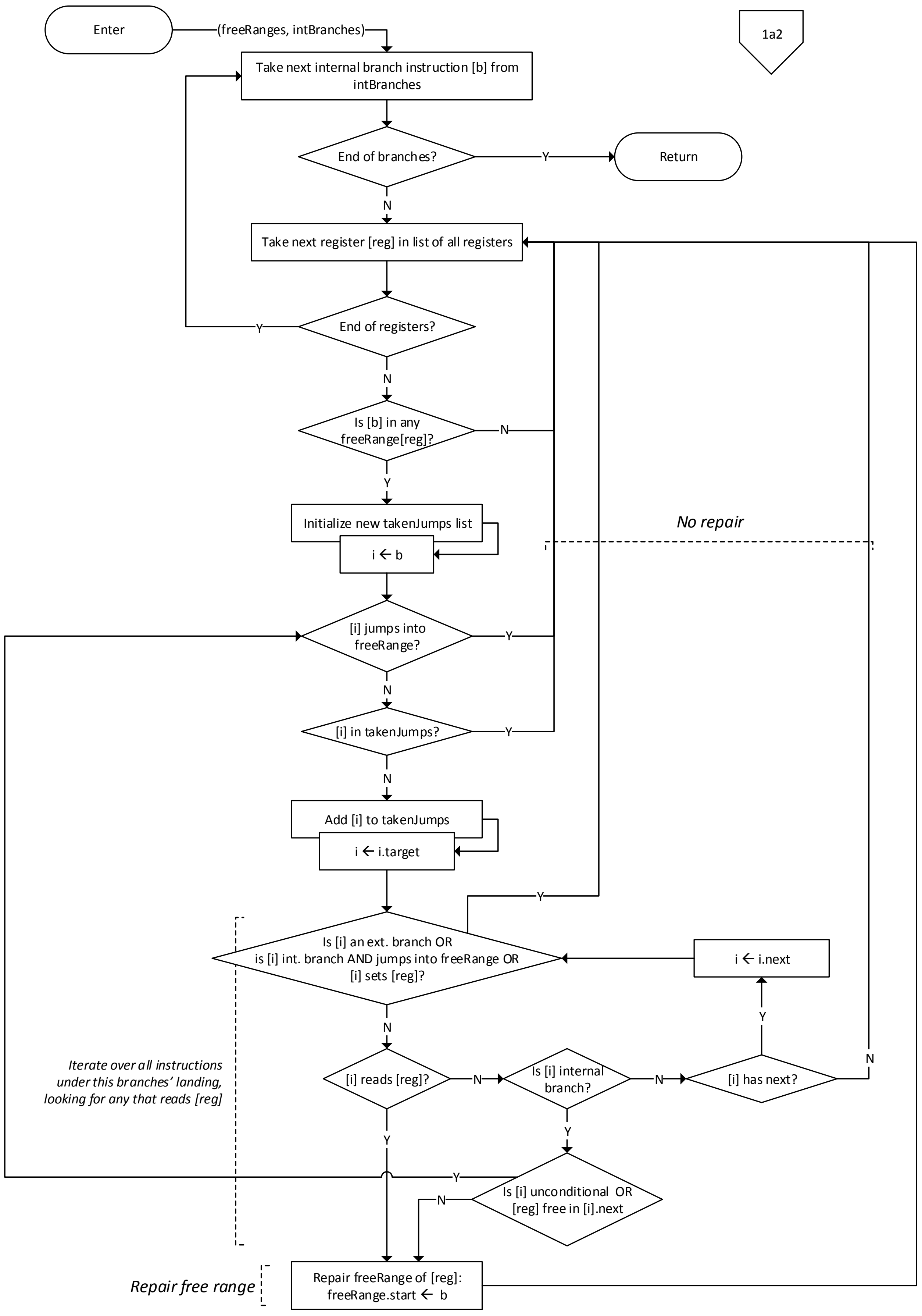
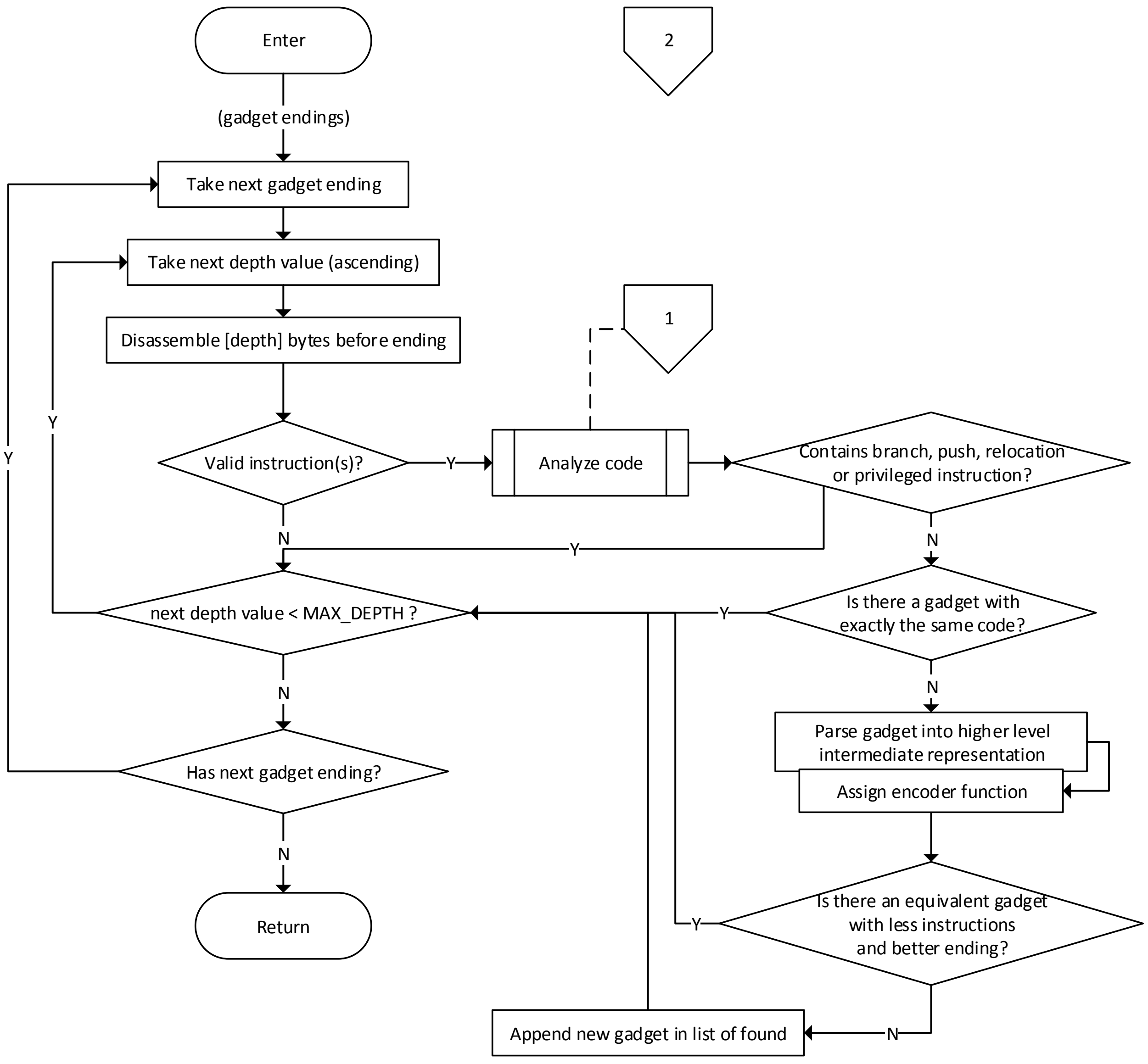

Enter
(freeRanges, intBranches)
1a2
Take next internal branch instruction [b] from intBranches
End of branches?
Y
Return
N
Take next register [reg] in list of all registers
End of registers?
Y
N
Is [b] in any freeRange[reg]?
N
Y
Initialize new takenJumps list
i ← b
No repair
[i] jumps into freeRange?
Y
N
[i] in takenJumps?
Y
N
Add [i] to takenJumps
i ← i.target
Y
Is [i] an ext. branch OR is [i] int. branch AND jumps into freeRange OR [i] sets [reg]?
i ← i.next
N
Y
Iterate over all instructions under this branches' landing, looking for any that reads [reg]
[i] reads [reg]?
N
Is [i] internal branch?
N
[i] has next?
N
Y
Y
Y
Is [i] unconditional OR [reg] free in [i].next
N
Repair free range
Repair freeRange of [reg]: freeRange.start ← b

Figure 8: Flowchart 1a2, calculation of free register ranges including internal branches

4.2.3 Gadget parsing and compilation to ROP

Figure 9 (flowchart 2) depicts the process of backwards searching for gadgets given the list of gadget endings found. The gadget ending finding process is omitted for the sake of brevity since it is relatively straight-forward.

“Analyze code” in this case refers to the same reverse analysis process described in paragraph 3.3 and flowchart 1 (Figure 5). The typical value of MAX_DEPTH is 20 bytes.



Enter
2
(gadget endings)
Take next gadget ending
Take next depth value (ascending)
Disassemble [depth] bytes before ending
1
Valid instruction(s)?
Y
Analyze code
Contains branch, push, relocation or privileged instruction?
N
Y
N
next depth value < MAX_DEPTH ?
Y
Is there a gadget with exactly the same code?
N
Parse gadget into higher level intermediate representation
Assign encoder function
Has next gadget ending?
N
Return
Y
Is there an equivalent gadget with less instructions and better ending?
N
Append new gadget in list of found
Y
Y

Figure 9: Flowchart 2, Backwards parsing of gadgets given the candidate gadget endings

Last, but certainly not least, Figure 10 (flowchart 3) documents the injection of gadgets and compilation to ROP given the source shellcode and the list of found gadgets. The complexity of the resulting code is estimated on a per-gadget basis in order to select the least complex encoding. This complexity estimation *c(g, i)* of gadget *g* encoding *i*, is calculated as:

$$c(g, i) = \text{number of VAs in the resulting chain} \times \text{number of instructions in } g$$

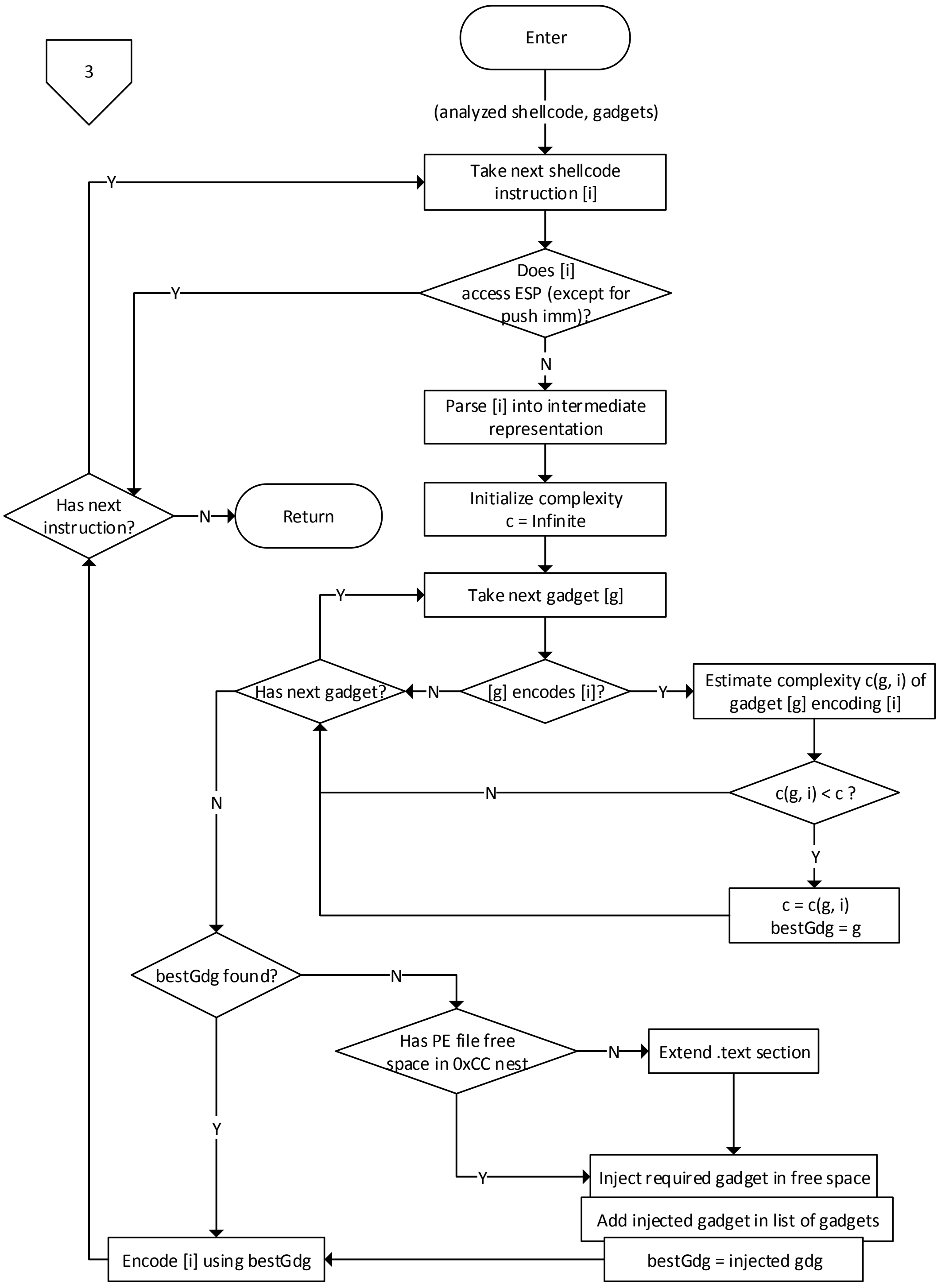

3
Enter
(analyzed shellcode, gadgets)
Take next shellcode instruction [i]
Does [i] access ESP (except for push imm)?
Y
N
Parse [i] into intermediate representation
Initialize complexity c = Infinite
Has next instruction?
Y
N
Return
Take next gadget [g]
[g] encodes [i]?
Y
N
Estimate complexity c(g, i) of gadget [g] encoding [i]
Has next gadget?
Y
N
c(g, i) < c ?
N
Y
c = c(g, i)
bestGdg = g
bestGdg found?
N
Y
Has PE file free space in 0xCC nest
N
Y
Extend .text section
Inject required gadget in free space
Add injected gadget in list of gadgets
bestGdg = injected gdg
Encode [i] using bestGdg

Figure 10: Flowchart 3, injection of gadgets and compilation to ROP

5. Results

In order to evaluate the proposed techniques we used the VirusTotal online antivirus scanning service [4] which at the time of this writing includes 57 AVs. For carrier PEs, we selected 9 32-bit executable of various sizes, most of them very popular and including certificates.

Table 7: List of PE files used as carriers for infection and validation

Executable	Size (KB)	Remarks	SHA256 hash
AcroRd32.exe	1489	Version 10.1.12 of Adobe Acrobat Reader X	a03297789b5a784af3765c523b33b9d54578e38a178ca67103b5e0e74f905331
cmd.exe	296	Version 6.1.7601.17514 Windows Command Processor	17f746d82695fa9b35493b41859d39d786d32b23a9d2e00f4011dec7a02402ae
Dummy.exe	56	An empty Win32 MFC application, straight from Microsoft Visual Studio 12	efd58b54923f2001c14065bc9687bdf345358ffce8914250b3be924f8bd37069
EssModel.exe	889	Version 2.2 of open source Eldean ESS-Model application	5b82358f54f21fdb0ea5a39c6d87b2cbfe730af4184e8cd0043cc1a4e098e6e8
firefox.exe	331	Version 35.0.0.5486 of Mozilla Firefox	11740f07a822637874da4eb4eafa309d145a1ca729779e30cb3d1e592c5484df
java.exe	172	Version 7.0.710.14 of Oracle Java	06889c037faab8379aaafb2bf9e77807e3d432da435cdab1244bff36c5c562d5
list_imports.exe	9	A tiny, Win32 console application written in C (compiled with MS compiler), listing all PE file imports	44b636ac9293df95eb42c8efe18b135649ebc2abc90ce9670d41ccc0b5c58e59
nam.exe	1829	Version 1.0a11a of “The Network Animator”	5d329bb39ba744cdba5e1afe107551c18ba0acd46cb6764391024a73aa2d583f
notepad++.exe	2348	Version 6.6.9.0 of the GNU text editor for windows	a11077cb6c209c67eb2d507d650fbee0925f3cbe860c70e0cd779b73f5af4b80

As source shellcode, we selected the most popular payloads of Metasploit [7]: Reverse TCP Shell, and Reverse TCP Meterpreter. For each PE and each shellcode we tried various combinations of encoding methods as listed in Table 8 resulting in a total of 126 samples.

Table 8: List of patching scenarios tested against VirusTotal

Code name	Description
Original	The file is not patched at all
ROP-Exit	The file is patched with the shellcode unrolled, converted to ROP, and entry point before the original program’s exit (hook ExitProcess or exit)
Unroll-Exit	The file is patched with the shellcode unrolled and entry point before the original program’s exit (hook ExitProcess or exit)
Exit	The file is patched with the shellcode intact and entry point before the original program’s exit (hook ExitProcess or exit)
ROP-d20	The file is patched with the shellcode converted to ROP, and entry point before the original program. An additional delay of 20 seconds is inserted before execution of the shellcode.
ROP	The file is patched with the shellcode converted to ROP, and entry point before the original program.
Shellcode	The file is patched with the shellcode intact, and entry point before the original program.

Figure 11 and Figure 12 depict the evasion ratios $(1 - \frac{positives}{total\ AVs})$ for each combination of method-PE file, for the reverse shell and the reverse meterpreter payloads respectively. In all cases, compiling to ROP and transferring control to the malware code before termination ("ROP-Exit" combination) produces the highest evasion ratio. In more than half of the test cases the method results in zero detections by AVs, while in the vast majority it results in less than two positives (an evasion ratio greater than 98.5%).

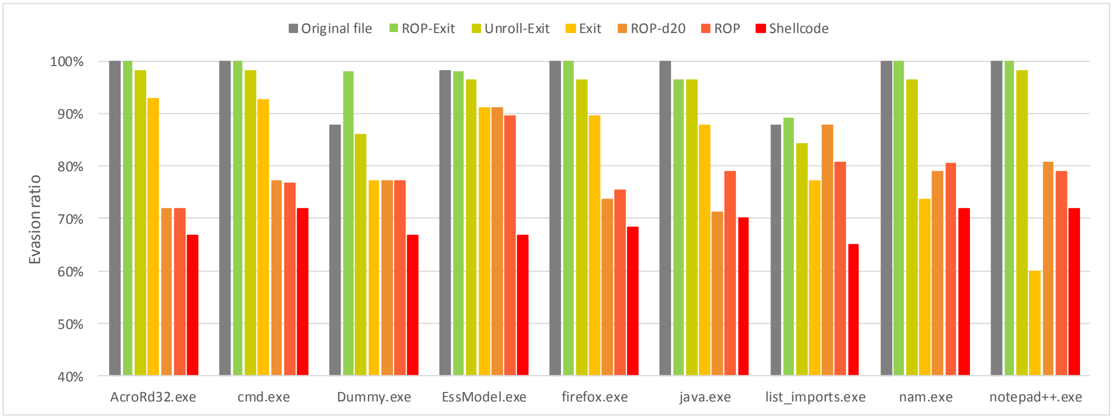


Figure 11: Evasion ratio for the reverse shell payload

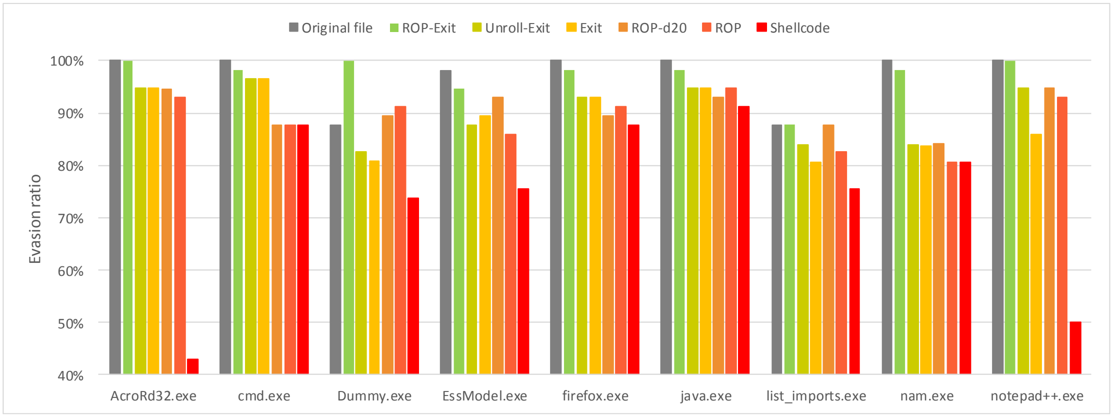


Figure 12: Evasion ratio for the reverse meterpreter payload

Figure 13 depicts the overall average evasion ratio for all the selected combinations of methods. ROP with entry point during termination ("ROP-Exit" case) produces on average slightly less positives (97.6% evasion rate) than the original file (97.1% evasion rate). This unexpected result is attributed to the increased false positives of the AVs for Dummy.exe and list_imports.exe which were falsely recognized as `Gen:Variant.Graftor.122324` and `Gen:Variant.Zusy.122107` respectively by the same 7 AVs (out of a total of 57).

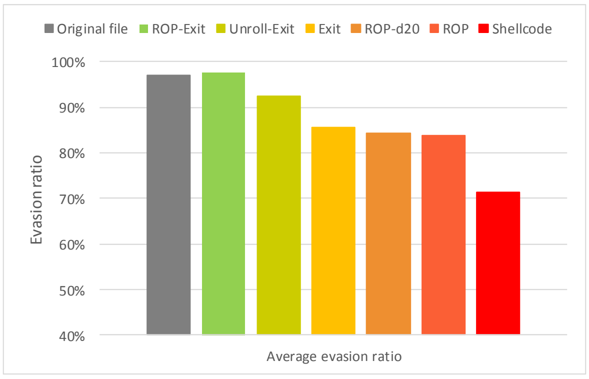


Figure 13: Average evasion ratio per combination of methods

We also notice that the second best results (in terms of higher evasion ratio) come from "Unroll-Exit" and the third best by "Exit" alone, while "ROP-d20" and "ROP" take two of the worst places, which lead us to the following conclusions:

- Behavioral analysis is equally important to static signatures for most AVs (from the ones that were alarmed) and is mostly performed during entry of executables
- Delaying execution of the shellcode has almost no effect which leads us to believe that probably some form of emu/simulation of the code is performed to gather the sequence of Win32 calls rather than pure virtualization/sandboxing and runtime hooking of Win32 API

Another suspicion is that many AVs rely (by a significant amount) on code statistics (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, there is high chance that they will be heavily affected by the ratio of patch size over total code size. For this reason we plot the average evasion ratio of the reverse shell and meterpreter payloads against the file size of all PEs in Figure 14.

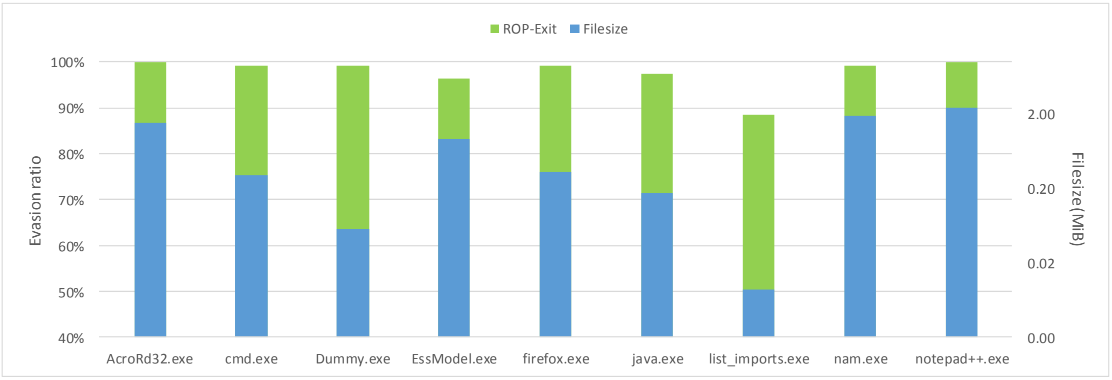


Figure 14: Average evasion ratio for "ROP-Exit" vs PE file size

The conjecture is confirmed by most cases (the two metrics seem correlated) except for instance by Dummy.exe versus EssModel.exe in which the bigger file (EssModel.exe) gets a lower evasion ratio.

A comparison is also made with Shellter v2.2 [1] and PEinject [2] in Figure 15. Shellter was used with its default options (i.e. with polymorphic junk code). In all cases in the Reverse TCP Shell was patched in. "ROP-Exit" has the highest evasion ratio in all tests. However, since Shellter does not encode the payload (neither does it claim to) the "Exit" scenario is also given to compare on the patching method only. The results are mixed between Shellter and

"Exit" case, with 5/9 higher evasion rates by "Exit", whereas in 8/9 tests "Exit" achieves higher rates than that of PEinject.

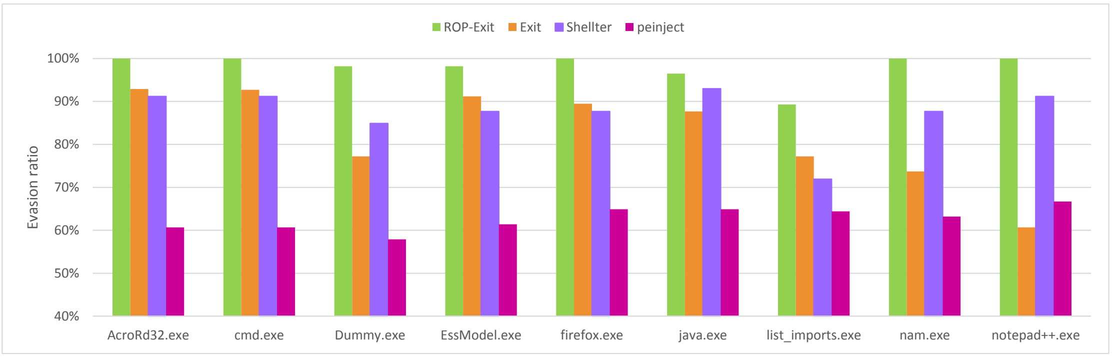


Figure 15: Comparison of evasion ratio between "ROP-Exit", "Exit" cases, Shellter and PEinject

Besides VirusTotal, all patched PEs were also tested against NCCGroup's "Experimental Windows .text section Patch Detector" [9]. This detector compares the executable sections in memory against the ones on disk to detect modifications/patching. Therefore, none was detected as patched, since our ROP patcher does not alter the .text section in memory (neither does it require to).

6. Conclusions

Most antivirus software rely on string signatures and mild behavioral profiling detection mechanisms. By encoding malicious code into its return-oriented equivalent and even by performing elementary permutations (unrolling), the former can be bypassed in the vast majority of cases. 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 thesis, 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%.

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 is not straight-forward to design, practical to implement, or even realistic to propose. 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, pretty much like what was started by Microsoft back in 2001 [8] but did not flourish.

References

[1] Shellter, https://www.shellterproject.com, last accessed on February 15, 2015

[2] Injecting Shellcode into a Portable Executable(PE) using Python, http://www.debasish.in/2013/06/injecting-shellcode-into-portable.html, last accessed on February 15, 2015

[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 February 15, 2015

[5] ProView disassembler, http://www.woodmann.com/collaborative/tools/index.php/PVDasm_Disassembly_Core_Engine, last accessed on February 15, 2015

[6] x86 Opcode and Instruction reference, http://ref.x86asm.net, last accessed on February 15, 2015

[7] Metasploit, http://www.metasploit.com/, last accessed on February 15, 2015

[8] Default Deny All Applications, http://www.windowsecurity.com/articles-tutorials/authentication_and_encryption/Default-Deny-All-Applications-Part1.html, last accessed on February 15, 2015

[9] Experimental Windows .text section Patch Detector, https://github.com/nccgroup/WindowsPatchDetector, last accessed on February 15, 2015