RopSteg: Program Steganography with Return Oriented Programming

Kangjie Lu
Georgia Institute of Technology
kjlu@gatech.edu

Siyang Xiong
D'crypt Pte Ltd
xiong_siyang@d-crypt.com

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

ABSTRACT

Many software obfuscation techniques have been proposed to hide program instructions or logic and to make reverse engineering hard. In this paper, we introduce a new property in software obfuscation, namely *program steganography*, where certain instructions are "diffused" in others in such a way that they are non-existent until program execution. Program steganography does not raise suspicion in program analysis, and conforms to the $W \oplus X$ and mandatory code signing security mechanisms. We further implement RopSteg, a novel software obfuscation system, to provide (to a certain degree) program steganography using return-oriented programming. We apply RopSteg to eight Windows executables and evaluate the program steganography property in the corresponding obfuscated programs. Results show that RopSteg achieves program steganography with a small overhead in program size and execution time. RopSteg is the first attempt of driving return-oriented programming from the "dark side", i.e., using return-oriented programming in a non-attack application. We further discuss limitations of RopSteg in achieving program steganography.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

Keywords

Code obfuscation, watermarking, program steganography, return-oriented programming

1. INTRODUCTION

Many software program obfuscation techniques have been proposed to deliberately conceal various aspects of an executable to make reverse engineering hard [7, 11, 20, 15, 26]. These techniques are powerful in terms of their robustness, semantic-preservation, obscurity, resilience, stealth, and other properties [7, 11, 15]. However, most existing techniques raise suspicion in program analysis, and may violate the $W \oplus X$ or mandatory code signing security mechanisms [14].

Software watermarking [6], on the other hand, tries to embed a secret message into a cover program, which is very similar to the concept of steganography, the art and science of hiding information. Although both software obfuscation and software watermarking are forms of security through obscurity, a notable difference in them are 1) software obfuscation tries to *transform* something while software watermarking tries to *hide* something; 2) target of software obfuscation is usually executable code in the original program, while that of software watermarking is usually additional secret message (data) that is not part of the original program.

This paper introduces a new property in software obfuscation, which, in some sense, combines the ideas of traditional software obfuscation and watermarking, namely *program steganography*. Program steganography refers to an interesting property that part of the executable code is hidden. It differs from existing software obfuscation in that the instructions are hidden instead of being transformed. It also differs from existing software watermarking in that part of the executable code in the original program, instead of some additional information, is hidden.

We do not introduce program steganography just for the sake of combining the ideas of existing obfuscation and watermarking techniques. Program steganography has some specific and useful properties absent from existing approaches, out of which the most important ones being not attracting attention to itself, a well documented advantage of steganography over cryptography. Even the strongest existing software obfuscation, in which instructions are encrypted to make static analysis next to impossible (e.g., [26, 20]), cannot hide the existence of the instruction sequence and leaves suspicion to program analysis. Program steganography, on the other hand, completely hides the existence of certain instructions from static analysis and raises no suspicion or attention. This is also different from self-generating code where instructions to be executed are generated by the program itself on the fly, which violates the $W \oplus X$ security mechanism and is not suitable in mandatory code signing environments (e.g., iOS). Note that we do not include resistance to *dynamic* analysis as a necessary property of program steganography, as the hidden instructions are intended to be executed in a dynamic run.

One of the reasons why program steganography has not been widely studied is its difficulty. It was not clear how some instructions to be executed could be completely hidden in an executable. However, steganography has been well

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.
Copyright 2014 ACM 978-1-4503-2278-2/14/03 ...$15.00.
http://dx.doi.org/10.1145/2557547.2557572.

studied to embed secret into documents, images, audio and video files, and so the concept behind it and the techniques to achieve it are well known. The gap lies in the availability of a special technique to achieve program steganography. We propose that using return-oriented programming [3, 4, 17, 19, 12] could be a good way to close this gap.

Return-oriented programming (ROP) [19] has attracted a lot of research attention in the last few years. The idea of using unintended gadgets (unintended instruction sequences ending with `ret`) in ROP is to disassemble instructions from the middle of some intended instruction to perform attacks. Such instructions are "hidden" in the sense that they were not intended to exist when the binary was created, and therefore a disassembler would never pick them up. If we could *intentionally* "diffuse" some instructions into such an unintended form, they will be hidden and therefore non-existent until program execution.

We design and implement *RopSteg*, a novel software obfuscation technique that hides the existence of program instructions while providing (to a certain degree) program steganography. More specifically, RopSteg hides instructions from static analysis while conforming to the $W \oplus X$ and mandatory code signing security mechanisms with minimal suspicions raised. We additionally evaluate RopSteg by applying it to different Windows executables, including desktop application, server program, and malware. Results show that RopSteg achieves program steganography with a small overhead in program size and execution time.

Note that RopSteg applies the concept of ROP in a completely new way. Return-oriented programming had always been considered as an attacking technique in previous research, where an attacker locates and uses unintended gadgets found in a given and vulnerable program. RopSteg, on the other hand, turns instructions into unintended form and embeds the ROP code into a program. In this respect, RopSteg is the first proposal of driving return-oriented programming from the "dark side", i.e., using return-oriented programming in a non-attack application[1].

In summary, our paper makes the following contributions.

- Introducing a new property in software obfuscation, namely, program steganography, to hide the existence of program instructions.
- Designing and implementing RopSteg, a novel technique for providing program steganography using return-oriented programming.
- Evaluating RopSteg by applying it to different Windows programs.
- First proposal of driving return-oriented programming from the "dark side".

2. PROGRAM STEGANOGRAPHY AND RELATED WORK

In this section, we first introduce the new property in software obfuscation, namely, *program steganography.* Due to its close relationship with a few previously proposed concepts, we also present a summary of these related works as well as their differences from program steganography.

Given a piece of executable code, program steganography refers to the result of hiding part of the program's functionality and the corresponding instructions such that they are non-existent till the point of execution. Figure 1 shows a simple example of a program segment that exhibits program steganography. Static analysis of this code segment reveals 100% disassembled instructions (see bottom of Figure 1). These instructions are prepared by a normal compiler and can be easily disassembled. However, the exact same byte sequence could also be interpreted in a different way to obtain completely different instructions (see top of Figure 1). A normal disassembler would not be able to pick up these instructions. They are hidden till the point of execution.

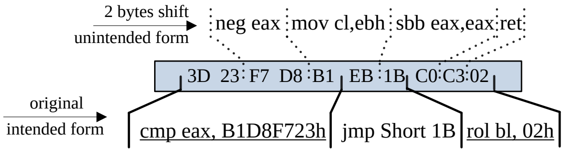


Figure 1: Program steganography example

Software watermarking.

Software watermarking is the closest to our property of program steganography. In software watermarking, some information is embedded in the software program to be reliably located and extracted [9, 8]. This is similar to program steganography in the sense that something is hidden in a program. The main difference is on the target of hiding — software watermarking usually hides a piece of information, e.g., the author's identity, which is a foreign object to the program; program steganography, on the other hand, hides some of its own instructions. The information hidden by software watermarking needs to be extracted via some special means, while instructions hidden by program steganography will be executed during program's normal runs.

Software obfuscation.

Many software program obfuscation techniques have been proposed to deliberately conceal various aspects of an executable to make reverse engineering hard. These techniques are powerful in terms of their robustness, semantic preservation, obscurity, resilience, stealth, etc [7, 11, 15, 26], most of which are designed to transform readable code into obfuscated code that is hard to understand or reverse engineer.

Two types of software obfuscation techniques are worth noting due to their close relationship with program steganography. One is to make disassembling of machine code difficult [11, 20, 15] so that only a small portion gets disassembled. Although in some sense program steganography also tries to hide information from the disassembler, it does it in such a way that the disassembler believes that it succeeds in disassembling 100% of the binary code, although the fact is that there are still instructions hidden.

Another work in software obfuscation technique applies encryption to some of the instructions so that they are hidden without a key. Encryption provides very strong protection; however, it always leaves suspicion to program analysis and attracts attention. Program steganography has a very similar objective, but tries to achieve it without attracting attention from the analyzer by hiding the existence of the instructions from static analysis. Program steganography could be considered a new property in software obfuscation.

[1] RopSteg could also be used by malware writers to hide malicious instructions to evade detection.

Self-generating code.

Executable code can be generated on the fly before they are executed, hiding themselves from static analysis of the program binary [20]. The difference between self-generating code and program steganography is subtle, in that self-generating code comes to live (e.g., code placed on writable and executable memory) before they are executed, while instructions hidden by program steganography never come to live. Even if program analysis is performed right before, during, or after the execution, the hidden instruction in program steganography never "exists" in any noticeable form and no self-modifying is introduced. This makes program steganography suitable in mandatory code signing environments, e.g., iOS, where self-generating code is not allowed.

Document, image, audio, and video steganography.

Steganography [2, 23] is a well studied technology widely applied to documents, images, audio, and video files. Steganography refers to hiding information in these files, while the information to be hidden is usually a foreign object, just like that in software watermarking. Program steganography, on the other hand, tries to hide part of itself.

Deniable encryption.

Analyzing the example in Figure 1, one may notice that the bytes in code segment are interpreted two times in two different ways, a concept similar to deniable encryption [16, 13]. Deniable encryption provides multiple ways of explaining a single ciphertext so that the creator of the message could deny having produced it. Inspired by deniable encryption, one way to provide program steganography is to prepare multiple ways of explaining some binary code, out of which one is given to the disassembler to make it believe that the code segment has been 100% disassembled, and others are used to execute the hidden instructions. Unfortunately, encryption is not a solution because we do not want to attract attention or to generate new code upon decryption.

Return-oriented programming.

Return-oriented programming (ROP) had been proposed as an attacking technique to perform arbitrary computation without injected code [3, 4, 5, 17]. The idea behind ROP is to disassemble the program under attack at different offsets and to execute the new instructions to perform arbitrary computation. This idea matches with the example shown in Figure 1 in that the hidden instructions could be those disassembled at a new offset. Our proposed system, RopSteg, actually applies the idea of ROP to achieve program steganography; see the next section for an overview.

3. OVERVIEW OF RopSteg

Having introduced the concept of program steganography, we now turn to our novel system RopSteg that provides the property of program steganography. RopSteg takes as input the binary instructions of a program P and some sequence of instructions I from P to be hidden. RopSteg tries to hide these instructions in a way that they are non-existent until being executed, i.e., I is hidden from static analysis but visible in a dynamic run. Note that as discussed in Section 2, RopSteg does not use encryption or dynamic code generation, and thus conforms to the $W \oplus X$ and mandatory code signing security mechanisms.

Figure 2 shows an overview of RopSteg and the four steps in which the obfuscated program executes. Code block 2 (ending at `addr2`) denotes I, which is the instruction sequence to be hidden. RopSteg modifies Code block 3 (starting at `addr1`) such that I is embedded in unintended form. RopSteg then replaces I with an *ROP board* which performs the control transfer. When the ROP code embedded at `addr1` finishes execution, control returns to `addr2`. To combat static analysis of finding values of `addr1` and `addr2`, RopSteg uses an *ROP generator* to dynamically calculate the values of them and to store them in memory.

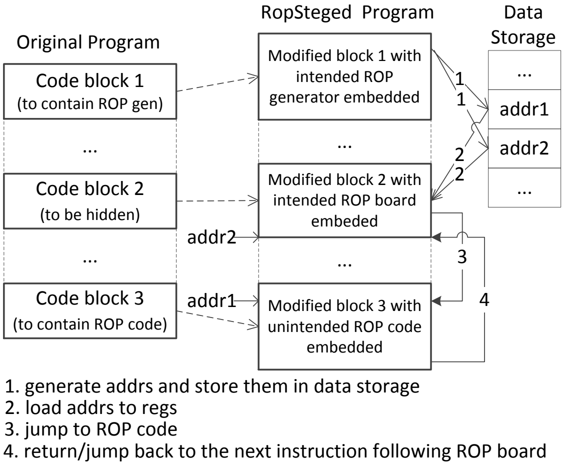

Original Program
RopSteged Program
Data Storage
Code block 1
(to contain ROP gen)
...
Code block 2
(to be hidden)
...
Code block 3
(to contain ROP code)
Modified block 1 with intended ROP generator embedded
...
Modified block 2 with intended ROP board embeded
...
Modified block 3 with unintended ROP code embedded
addr2
addr1
1
1
2
2
3
4
...
addr1
addr2
...
1. generate addrs and store them in data storage
2. load addrs to regs
3. jump to ROP code
4. return/jump back to the next instruction following ROP board

Figure 2: An overview of RopSteg

Figure 3 shows an example where I = `<neg eax; sbb eax, eax; ret>`[2]. RopSteg first finds an unintended form of I. This could be easy when I is short and when P is large, but it could also be impossible unless we insert additional instructions into P, as shown in Figure 3b at `addr1`. After the unintended form of I is located, RopSteg replaces the original I with an ROP board (see Figure 3c), which loads `addr1` into `eax`, stores `addr2` on the stack, and jumps to `addr1`. In the end, RopSteg inserts an ROP generator (see Figure 3d) to dynamically calculate and store `addr1` and `addr2`.

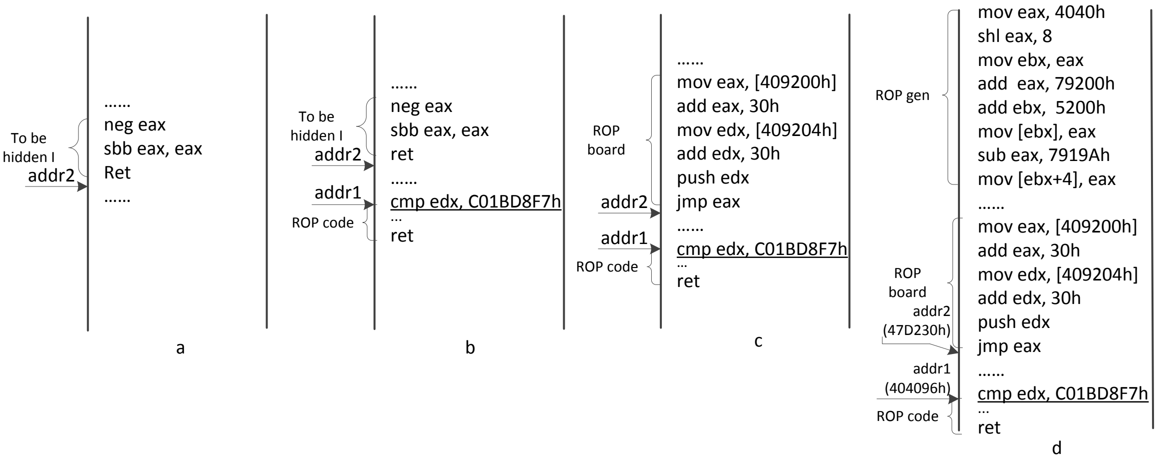
4. DESIGN OF RopSteg

As explained in Section 3, RopSteg performs three main modifications on P, namely the embedded I in the form of ROP code, the ROP board to facilitate control transfers, and the ROP generator to dynamically generate ROP gadgets. In this section, we present details of these three parts and outline the binary rewriting to perform the modification.

4.1 Finding and constructing unintended ROP code

Previous work on ROP has demonstrated that gadgets and unintended code can be found efficiently and automatically [19, 18]. However, RopSteg uses ROP in a different setting in which the execution of ROP is legitimate and planned. RopSteg could modify P to plant seeds for ROP execution. Therefore, the algorithm to find and construct unintended ROP code is different from any previous work.

[2] The return or return-like instruction was inserted at the end to facility a return after I finishes execution.



```
......
neg eax
sbb eax, eax
Ret
......
```

```
......
neg eax
sbb eax, eax
ret
......
cmp edx, C01BD8F7h
...
ret
```

```
......
mov eax, [409200h]
add eax, 30h
mov edx, [409204h]
add edx, 30h
push edx
jmp eax
......
cmp edx, C01BD8F7h
...
ret
```

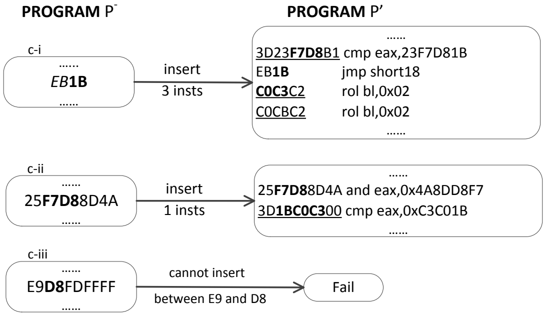
```
mov eax, 4040h
shl eax, 8
mov ebx, eax
add  eax, 79200h
add ebx,  5200h
mov [ebx], eax
sub eax, 7919Ah
mov [ebx+4], eax
......
mov eax, [409200h]
add eax, 30h
mov edx, [409204h]
add edx, 30h
push edx
jmp eax
......
cmp edx, C01BD8F7h
...
ret
```

Figure 3: Using ROP for program steganography

RopSteg first removes I from P to obtain P^-. It then uses a modified Galileo algorithm G to find a sequence of candidate instructions C that fully or partially match with I. For each $c \in C$ that does not fully match with I, RopSteg looks for I' that is semantically equivalent with I and P' that is semantically equivalent with P^- such that the resulting c' fully matches with I'.

4.1.1 A modified Galileo algorithm G

Unlike the original Galileo algorithm presented when return-oriented programming was first introduced [19], our modified algorithm G is flexible enough to be able to find *partial matches* of the unintended form of I. Refer to an example shown in Figure 4 c-i where I = <F7 D8 1B C0 C3>. Upon searching for I in P^-, we realized that no exact match exists, and therefore G output a partial match c = <EB 1B> (a one-byte match). G then inserts three instructions into P^- to form P' such that I' = <F7 D8 B1 EB 1B C0 C3> is semantically equivalent with I and has an exact match in P'.



```
......
EB1B
......
```

```
......
3D23F7D8B1 cmp eax,23F7D81B
EB1B       jmp short18
C0C3C2     rol bl,0x02
C0CBC2     rol bl,0x02
......
```

```
......
25F7D88D4A
......
```

```
......
25F7D88D4A and eax,0x4A8DD8F7
3D1BC0C300 cmp eax,0xC3C01B
......
```

```
......
E9D8FDFFFF
......
```

Figure 4: Partial matches found by G

It may sound simple, but G is much more complicated than a substring search to maximize matches. For example, G might consider a partial match c = <E9 D8 FD FF FF>. However, no matter how we insert additional instructions into P^- to produce a semantically equivalent P', the byte immediately proceeding the matching byte D8 can never change, which means that the resulting c' will always have E9 proceeding the matching byte D8. This happens because the matching byte D8 appears in the middle of an instruction instead of locating at the beginning as in c-i in Figure 4.

Note that here we only insert instructions when transforming from P^- to P' and from I to I' (additions). G rules out a candidate match c containing a matching byte b if

- $!\mathsf{isLast}(b, P^-) \cap !\mathsf{isLast}(b, I) \cap \lhd(b, P^-) \neq \lhd(b, I)$, and;
- $!\mathsf{isFirst}(b, P^-) \cap !\mathsf{isFirst}(b, I) \cap \rhd(b, P^-) \neq \rhd(b, I)$.

where $\mathsf{isFirst}(b, X)$ and $\mathsf{isLast}(b, X)$ denote that b is the first or the last byte in the corresponding instruction in context X, respectively; and $\lhd(b, X)$ and $\rhd(b, X)$ denote the byte proceeding or following b in context X, respectively. In Figure 4, c-iii is filtered out because we cannot insert bytes between E9 and D8. After finding the valid candidates, RopSteg arranges them in a sequence C according to the following considerations (in order of importance).

1. number of instructions c covers;
2. number of matching bytes c covers;
3. number of matching bytes that satisfy $\mathsf{isFirst}(b, P^-)$;
4. number of matching bytes that satisfy $!\mathsf{isFirst}(b, P^-) \cap !\mathsf{isLast}(b, P^-)$;
5. number of matching bytes that satisfy $\mathsf{isLast}(b, P^-)$.

It it easier to construct I' when b is the first byte in an instruction in P^-, and that is why the last three counts are in the order in which they are presented above. We explain this in more detail in the next subsection. In the examples presented in Figure 4, c-ii ranks higher than c-i because c-ii matches one more byte than c-i. With all the candidate matches found and arranged in a ranking sequence, RopSteg proceeds to construct I' and P' such that there exists a corresponding c' that fully matches I'.

4.1.2 Constructing equivalent versions of I *by inserting ineffective instructions*

RopSteg constructs semantically equivalent P' and I' by inserting *ineffective instructions*, which have no effect in the semantics of the corresponding execution context.

Context Insensitive	Context Sensitive	
Instructions	Instructions	Context
mov edi, edi	add eax,???? sub ebx,????	CF
or eax, eax	cmp ebx,????	CF/ZF/SF/OF/AF/PF
push ebp; pop ebp	test eax,????	CF/ZF/SF/OF/AF/PF
xchg eax, ebx; xchg eax, ebx	mov eax,????	eax not in use

Table 1: Two types of ineffective instructions

Ineffective instructions.

RopSteg uses two types of ineffective instructions, context insensitive ones and context sensitive ones. Table 1 shows some examples of ineffective instructions RopSteg uses (where `?` denotes a "don't care" byte).

Ineffective instructions that are context insensitive never change anything regardless of the execution context, e.g., `<mov eax,eax>`. On the other hand, context sensitive ones only possess the ineffectiveness property in some particular execution context, e.g., `<add eax,????; sub eax,????>` is ineffective when flag `CF` is not in use.

The use of ineffective instructions is a well-studied area, and they have been widely used in previous research work to produce polymorphic and metamorphic malware [21, 10, 1]. RopSteg takes advantage of the existing work in this area and constructs a database of ineffective instructions, which currently contains 230 entries (still evolving though). It is our future work to explore other types of ineffective instructions, and to experiment inserting more context sensitive ones to P^-. Expanding the search space of ineffective instruction will result in higher success rates of RopSteg in providing program steganography.

Inserting ineffective instructions into P^- *.*

Inserting ineffective instructions into P^- is one of the most complicated tasks in RopSteg. Here we first discuss the sensitive context check, and then explain the different types of instructions to be inserted.

RopSteg performs the context sensitive analysis with the classical *def-use* chain analysis [24]. First, we delineate the context from the insertion point to the end of the function (taking direct jumps into consideration). If there is any use of the resources (e.g., the flags) or an indirect jump instruction before a re-def of the resources, we consider the resources is sensitive in the context; otherwise insensitive.

To explain the different types of ineffective instructions to be inserted, we consider three cases shown in Figure 5 where $\mathsf{I} =$ `F7 D8 1B C0 C3`, i.e., `<neg eax; sbb eax, eax; ret>`.

The first case is when a matched byte is the first in an instruction in P^- before which an unmatched byte exists, i.e., $\lhd(\mathsf{b}, \mathsf{P}^-) \neq \lhd(\mathsf{b}, \mathsf{I}) \,\cap\, \mathsf{isFirst}(\mathsf{b}, \mathsf{P}^-)$. To increase the matching between P^- and I, we insert an instruction right before the matched one in P^- where the last byte of the instruction inserted is $\lhd(\mathsf{b}, \mathsf{I})$. As shown in case 1 in Figure 5, `C3` is the original matching byte. RopSteg inserts an instruction ending with `<F7 D8 1B>` to increase matching with I.

In this first case, we always manage to find such an instruction from our database of ineffective instructions. The catch here is that our requirement on the instruction is at its last (few) byte(s), and there exists ineffective instructions where the last few bytes are "don't cares". With our def-use chain analysis, RopSteg selects (context insensitive or sensitive) ineffective instructions, e.g., `mov ebx, ????`, where `ebx` is not in use in the context. Note that this case also corresponds to the third criteria when ordering the candidate matches (see Section 4.1.1).

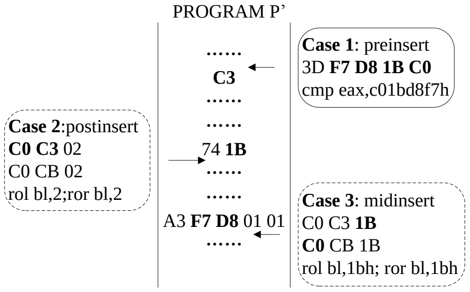


Figure 5: Locations of the matching/unmatching bytes in `c`

The second case is an exact opposite, i.e., when a matched byte is the last byte in an instruction in P^-, after which an unmatched byte exists ($\rhd(\mathsf{b}, \mathsf{P}^-) \neq \rhd(\mathsf{b}, \mathsf{I}) \,\cap\, \mathsf{isLast}(\mathsf{b}, \mathsf{P}^-)$). In this case, RopSteg tries to insert an ineffective instruction right after the matched instruction; however, such an ineffective instruction is the most difficult to find because its first byte is given as $\rhd(\mathsf{b}, \mathsf{I})$ (e.g., `C0` in case 2 in Figure 5). The database of ineffective instructions we have right now does not contain one for every possible initial byte, and therefore the success rate in this case is about 36% (our ineffective instruction database covers about 36% opcode). Note that this case corresponds to the fifth criteria when ordering the candidate matches (see Section 4.1.1).

The third case deals with a more general scenario where a matched byte appears in the middle of an instruction ($\rhd(\mathsf{b}, \mathsf{P}^-) \neq \rhd(\mathsf{b}, \mathsf{I}) \,\cap\, !\mathsf{isFirst}(\mathsf{b}, \mathsf{P}^-) \,\cap\, !\mathsf{isLast}(\mathsf{b}, \mathsf{P}^-)$). The successful rate of this case varies a lot depending on the actual scenario, and it usually takes the longest time to search for an ineffective instruction. That is why we give it the second lowest ranking as discussed in Section 4.1.1. In case 3 of Figure 5, we could not find a single ineffective instruction that meets all the requirements.

We also exercise care during the insertion process to minimize suspicions for steganalysis. For example, we only insert ineffective instructions that are commonly used in normal programs (e.g., `jecxz` is not used), we insert additional instructions to make instruction sequences look normal (e.g., instead of having `<push ebp; pop ebp>`, we change it to `<push ebp; pop eax; mov ebp, eax>`). The insertions usually result in additional unmatched bytes in I and `c`. We perform additional check to make sure the newly inserted unmatched bytes constitute an ineffective instruction in I.

4.2 ROP Board

After constructing I' and its unintended form, the next step is to transfer control to the unintended gadgets. As shown in Figure 3, ROP board performs this assuming `addr1` (address of the unintended code) and `addr2` (address of the instruction after the original I) are already stored in memory (what the ROP generator is supposed to do, see Section 4.3). To make it difficult for static analysis to detect the ROP board, RopSteg has multiple ways of accessing data storage in order to load them into registers. For example, the address of memory storage is indirectly calculated rather than an immediate as shown in Figure 3. After loading the addresses into registers, ROP board makes an indirect jump to `addr1`. `addr2` might be loaded onto the stack if the ROP code ends with a `ret` instead of an indirect jump.

4.3 ROP Generator

With I' constructed and ROP board connecting P' with I', RopSteg manages to hide I in unintended form. However, `addr1` and `addr2` could raise suspicion in steganalysis; therefore, RopSteg introduces ROP generator to make such analysis difficult. As shown in Figure 3, ROP generator dynamically calculates the address of the ROP code and the address to which ROP code returns, and stores them in the data segment. In the ASLR environment, the address of data segment is randomized. RopSteg adopts a common way, i.e., `call-pop` instruction sequence, used in position-independent executable (`PIE`) to load the value of the current `eip` into `eax`. By adding the offset to `eax`, we can get the randomized address of the data segment.

4.4 Binary Rewriting

After successfully constructing I and connecting I' and its original context, RopSteg constructs the new binary via binary rewriting. Although we envision that RopSteg could be integrated with a compiler to produce the new executable from source code directly, for the purpose of finer-grained performance evaluation, we implemented RopSteg as a standalone component using binary rewriting in the current version and leave the integration with a compiler as future work.

As shown in Figure 3, existing code section needs to be expanded to make room for new instructions. Operands of jump and call instructions, direct or relative, need to be relocated. When the expansion goes beyond the original code section, other sections after it have to be moved backward.

If the program is ASLR-enabled with relocation tables, the binary rewriting would be much easier as the positions of the code blocks could be changed by simply adjusting the addresses in direct call/jump instructions and modifying the entries in the relocation table. Similarly, in a PIE-enabled program, we can simply identify the program counter related instructions and adjust the relative addresses in them to address the offset problem. The binary rewriting for these two scenarios is easy and we will not elaborate it here. When the program strips the relocation information or disables PIE, further adjustments are required to the following code, which is challenging.

In order to handle such case, accurate disassembling would be the most important part. We use IDA Pro (together with NDISASM[3]) to obtain function boundary information and the set of potential targets of branch instructions. And then we use *binary sled*, inspired by binary stirring [25], to address the offset problem of binary rewriting. The basic idea in binary stirring is to put the extended blocks in a new `.text` segment. When control is transferred to the original code (with lookup table), it redirects control to the corresponding code in the new `.text` segment.

In order not to raise suspicion for steganalysis, we perform three additional steps. First, our inserted code block could be put either in the old `.text` or the new one. Second, RopSteg gathers the "innocent blocks" that are relatively large (e.g., greater than 100 bytes) but don't contain any jump instructions (the locality of jump instruction might raise suspicions) and put them in the new `.text` segment. After that, the original innocent blocks in the old `.text` can be used as a container to put our inserted code. Lastly, the indirect binary sled makes use of indirect jump/call instructions to transfer control to the corresponding block in the new `.text` segment. In this way, every code block could potentially be the block that contains our inserted code, and steganalysis is difficult.

5. EXPERIMENTS AND EVALUATION

In this section, we evaluate the effectiveness of RopSteg in providing program steganography in a few scenarios, one in hiding a secret algorithm, one in hiding some malicious code, and others in hiding random instructions. In addition, we analyze the overhead of the resulting binary in both program size and execution time.

Our experiments were performed with Microsoft Windows 7 ultimate on a desktop computer with AMD Phenom II X6 1090T CPU at 3.21GHz and 4GB of RAM. We implement RopSteg in C/C++ with around 7000 LOC.

5.1 Experiments

5.1.1 *Protecting a secret algorithm*

In this experiment, we apply RopSteg to hide the quicksort algorithm in `searchcand.exe` (a program we developed as part of RopSteg to search for unintended gadgets).

The original quick sort in `searchcand.exe` corresponds to 90 instructions or 266 bytes. We choose five critical instruction sequences (core variable calculation, control flow prediction, function calls, etc.) each containing one to three instructions for hiding, see Figure 6.

5.1.2 *Hiding malicious code*

Another use of RopSteg could be to hide malicious code. We select `trojan.dll` (the main malicious module of malware `Gh0st`) as the example, whose functionality includes FileManagement, ScreenMonitor, KeyMonitor, RemoteShell, and SystemManager, and use RopSteg to hide virus signatures in it. The signature of `trojan.dll` was located with MyCCL (a tool used to identify malicious feature) and is shown in Table 2[4].

Location	Instructions	Machine code
0x000000DB	mov eax,[ebp-118h]; push eax	8B85E8FEFFFF 50
0x00000CA4	mov edx,[ebp+8]; call [edx]	8B5508 FF12
0x00000800	and [esi+0Dh], 0EEh;	80660DEE

0x00006980	mov edx,[ebp+8];
	mov eax,[ecx*4+edx+4];
	add eax, 1
		8B5508
		8B448A04
		83C010

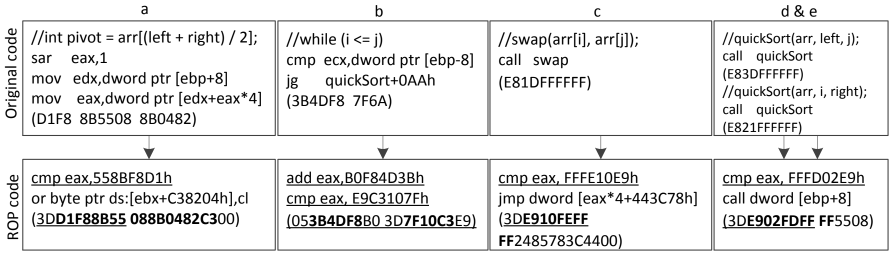
Table 2: Signature of trojan

5.1.3 *Hiding random instruction sequences*

We pick six more common `x86` Windows programs (see Table 3) and randomly select 100 different instruction sequences in each of the eight programs to hide. The instruc-

[3] `http://www.nasm.us/doc/nasmdoca.html`

[4] Here we focus on the signature in code section only and ignore that in the data segment.



```
//int pivot = arr[(left + right) / 2];
sar    eax,1
mov   edx,dword ptr [ebp+8]
mov    eax,dword ptr [edx+eax*4]
(D1F8  8B5508  8B0482)
```

```
cmp eax,558BF8D1h
or byte ptr ds:[ebx+C38204h],cl
(3DD1F88B55 088B0482C300)
```

```
//while (i <= j)
cmp  ecx,dword ptr [ebp-8]
jg      quickSort+0AAh
(3B4DF8  7F6A)
```

```
add eax,B0F84D3Bh
cmp eax, E9C3107Fh
(053B4DF8B0 3D7F10C3E9)
```

```
//swap(arr[i], arr[j]);
call   swap
(E81DFFFFFF)
```

```
cmp eax, FFFE10E9h
jmp dword [eax*4+443C78h]
(3DE910FEFF
FF2485783C4400)
```

```
//quickSort(arr, left, j);
call   quickSort
(E83DFFFFFF)
//quickSort(arr, i, right);
call   quickSort
(E821FFFFFF)
```

```
cmp eax, FFFD02E9h
call dword [ebp+8]
(3DE902FDFF FF5508)
```

Figure 6: RopSteg on quick-sort

tion sequences ranges from 2 to 13 bytes in length and cover different types of instructions including load/store, arithmetic, conditional branch, function call, system call, etc. Results are shown in 5.2.

5.2 Results and evaluation

All instruction sequences were successfully hidden by RopSteg. We use a linear sweep disassembler, `objdump` (Windows version), and a recursive disassembler, IDA pro, to disassemble the obfuscated programs. Neither could locate any of the hidden instruction sequences.

For the experiment of Section 5.1.2, we use Kaspersky, Macfee, and 360 Anti-Virus to scan the resulting binaries. Results show that none of them could identify any of the signatures. This also confirms our intuition that ROP generator and ROP board use very common instructions (e.g., `load`, `store`, and arithmetic operations) which, at least, does not raise suspicions on existing anti-virus engines.

In the following sections, we show more detailed results and our analysis on the results of these experiments.

5.2.1 Short I

One interesting finding is that the shorter I is, the more likely RopSteg succeeds in hiding it. This is intuitive as the shorter I is, the more likely that G finds relatively long candidate matches, making it easier to find P′ and I′.

Figure 6 shows the five instruction sequences from `searchcand.exe` that were successfully hidden and their corresponding ROP code in unintended form (bold face). We also show the corresponding machine code in square brackets. Instructions underlined are ineffective instructions RopSteg inserts.

It shows that longer I (cases A and B) results in fewer matching candidates (less than 100 for case A and zero for case B), while short ones (cases C, D, and E) result in more than 500 matching candidates. Therefore, a useful strategy RopSteg uses is to divide I into short sequences (usually fewer than 5 bytes long) that only have one or two instructions to obtain high succeed rate (100% in our experiments).

5.2.2 Size and runtime overhead

The insertion of ROP generator, ROP board, and ROP code are the main contributors to the increase in program size, which usually add about 30 bytes, 25 bytes, and fewer than 10 bytes, respectively. Table 3 presents the increase in size of the program when 100 instruction sequences are hidden (without reusing the ROP generator). We find that the increase in bytes remains more or less a constant, which translates to a small percentage for relatively large programs (except one in our experiments).

Program	Original size	Size increment	
	(bytes)	(bytes)	(percentage)
AcroRd32.exe	806941	5613	0.70%
iexplore.exe	13129	6410	48.80 %
java.exe	93240	5836	6.26%
calc.exe	75440	5890	7.80%
ftpsvc2.dll	100475	5717	5.69%
trojan.dll	100864	5712	5.66%
searchcand.exe	1335472	5507	0.41%
shell32.dll	2075888	4887	0.24%

Table 3: Size increment

We monitor the overhead of specific operations performed by `trojan.dll` to get an idea of the runtime overhead. Table 4 shows the time to execute five different operations (average taken on 100 runs) before and after applying RopSteg. We notice that the runtime overhead resulted from our modification to the Trojan is small.

Operation	Original (ms)	Modified (ms)
FileManagement	5313	5344
ScreenMonitor	62	62
KeyMonitor	31	40
RemoteShell	281	319
SystemManager	78	94

Table 4: Runtime overhead

6. CONCLUSION AND LIMITATION

In this paper, we design and implement RopSteg to make program steganography using return-oriented programming. We show that RopSteg successfully hide program instructions such that they are hidden from static analysis. Here we discuss the potential limitations of RopSteg.

Dynamic analysis.

As discussed in Section 1, program steganography does not intend to hide instructions from dynamic analysis. The hidden instructions will eventually get executed, and dynamic analysis could reveal their existence. There have been proposed methods [22, 20] to resist dynamic analysis, and we consider combining them with RopSteg a potential future direction of our research.

Compatibility with ROP defenses.

In the past several years, many ROP defense and detection techniques have been proposed. As RopSteg makes use of ROP to achieve steganography, it is incompatible with ROP defense and could be detected as malicious at run time. We argue that this is mainly due to the fact that ROP has always been considered in the "dark side" in the literature, which is no longer true with the introduction of RopSteg. Future ROP defenses would then need to carefully differentiate between ROP in attack and ROP in program steganography.

7. REFERENCES

[1] K. G. Anagnostakis and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *In USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[2] D. C. B. Anckaert, B. De Sutter and K. D. Bosschere. Steganography for executables and code transformation signatures. *Lecture Notes in Computer Science*, pages 425–439, 2005.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS 2008)*, Alexandria, VA, USA, Oct.27-31, 2008.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS 2010)*, Chicago, IL, USA, Oct 4-8, 2010.

[5] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security the case of return-oriented programming and the avc advantage. In *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE09)*, Montreal, Canada, Aug. 10-11, 2009.

[6] C. T. Christian Collberg. Software watermarking: models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 99)*, San Antonio, Texas, USA, Jan. 20-22, 1999.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. 1997.

[8] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28:735–746, 2002.

[9] P. COUSOT and R. COUSOT. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2004)*, 2004.

[10] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. V. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack magazine, 9(61)*, Aug. 2003. `http://www.phrack.org/issues.html?issue=61&id=9`.

[11] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of 10ththe ACM Conference on Computer and Communications Security (CCS 2003)*, Washington, DC, USA, Oct. 27-30,2003.

[12] K. Lu, D. Zou, W. Wen, and D. Gao. Packed, printable, and polymorphic return-oriented programming. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011)*, Menlo Park, California, USA, September 2011.

[13] M. K. M. Klonowski, P. Kubiak. Practical deniable encryption. *LNCS, Springer*, 4910:599–609, 2008.

[14] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. Wiley, May 8, 2012.

[15] I. V. Popov, S. K. Debray, and G. R. Andrews. bi obfuscation using signals. In *Proceedings of the 16th USENIX Security Symposium (Security 2007)*, Boston, MA, USA, Aug 6-10,2007.

[16] M. N. R. Canetti, C. Dwork and R. Ostrovsky. Deniable encryption. In *Proceedings of the 17th Annual International Cryptology Conference (CRYPTO 1997)*, Santa Barbara, California, USA, August, 1997.

[17] R. Roemer, E. Buchanan, H. Shacham, and S. Savagm. Return-oriented programming: Systems,languages, and applications, 2009.

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In *Proceedings of the 20th USENIX conference on Security (Security'11)*, San Francisco, CA, USA, August, 2011.

[19] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, USA, Oct. 29-Nov. 2,2007.

[20] M. Sharif, A. Lanzi, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS 2008)*, San Diego, CA, USA, Feb. 8-11, 2008.

[21] F. Skulason. 1260-the variable virus. 1990. Virus Bulletin.

[22] C. Song, P. Royal, and W. Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proceedings of The 7th USENIX conference on Hot topics in Security (HotSec 2012)*, Bellevue, WA, USA, August 2012.

[23] G. R. Tadiparthi and T. Sueyoshi. A novel steganographic algorithm using animations as cover. *Information Technology and Systems in the Internet-Era*, 45:937–948, Nov, 2008.

[24] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[25] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS 2012)*, Raleigh, NC, USA, Oct, 2012.

[26] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS 2010)*, Chicago, IL, USA, Oct 4-8, 2010.