Leave It to Weaver

William Mahoney Joseph Franco Greg Hoff
University of Nebraska at Omaha
6001 Dodge Street
Omaha, Nebraska 68182
{ wmahoney, jfranco, ghoff } @unomaha.edu

J. Todd McDonald
University of South Alabama
1121 Shelby Hall
150 Jaguar Drive
Mobile, AL 36688
jtmcdonald@southalabama.edu

ABSTRACT

Malware authors make use of several techniques to obfuscate code from reverse engineering tools such as IdaPro. Typically, these techniques tend to be effective for about three to six instructions, but eventually the tools can properly disassemble the remaining code once the tool is again synchronized with the operation codes. But this loss of synchronization can be used to hide information within the instructions – steganography. Our research explores an approach to this by presenting "Weaver", a framework for executable steganography. "Weaver" differs from other techniques in how it hides malicious instructions: the hiding instructions are prepared by generating an assembly listing of the program and finding candidate hiding locations, the steganography instructions are prepared by creating an assembly listing of the program to obtain the operation codes to be hidden, and the "weaving" process merges the two. This "weaving" attempts to place all the steganography instructions into candidate locations found in the hiding instructions.

CCS Concepts

• **Security and Privacy → Software and Application security**
• **Applied Computing → Computer Forensics**

Keywords

Steganography; Reverse engineering; Information retrieval

1. INTRODUCTION

When data is encrypted, users of the data, or others viewing that data, can use entropy as a means to detect the presence of a private message. Without knowledge of the key, data cannot be read, but nonetheless an observer can reasonably know a message is present. Steganography, on the other hand, hides data in plain sight, such as inside a digital image file as part of its bit structure. An outsider viewing the picture may not know that the image contains hidden data; it looks like just a typical harmless image. The process of steganography ("stego" for short) is valuable where sending encrypted messages might raise suspicion, such as in a foreign country which suppresses free speech. Steganography might also be used for intellectual property protection, for example by "watermarking" the ownership of a digital artifact in such a way that one can prove the origin of that artifact.

SSPREW-8, December 3–4, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6096-8/18/12…$15.00
https://doi.org/10.1145/3289239.3291459

Many off-the-shelf techniques are readily available for embedding digital data, particularly in image files. A simple technique is to change the colorization of a photo by manipulating the colors in such a minimal way that is too subtle for a notable difference to be seen. This method might change the least significant bit in a color, distributing the hidden data one bit at a time over the image. Of course, changing the least significant bit of a color may not cause the image to change much, whereas changing the least significant bit of an ASCII character has a much larger impact; the media is different. What might the hidden data consist of? Anything; it might be a secret message to be delivered, a second picture hidden inside the first, or an executable program, delivered by digital photo. Embedding executable code in graphic images [36], for example, has become a natural extension of hiding arbitrary data [32,33,34,37]. If, using steganography, a picture can hide executable software, then it is reasonable to think that one might use steganography to hide a picture inside of executable code. And for that matter, one should be able to hide executable code inside of other executable code.

This is the subject of our work here, which we call executable code weaving. We restrict our study specifically to x86-64 since as detailed below, variable length instructions are critical for our process. With our method, we describe how one executable program can be hidden inside another by replacing certain operands in the hiding program with operation codes from the hidden payload. Our approach necessitates compiling to assembly language, examining the output files of the assembler, and then merging the two files into one. We demonstrate the feasibility of the method by showing an example of a hashing function which includes a payload capable of starting a Linux command prompt. This example is worked through step by step and results from disassemblers show that the payload is not normally detected.

In section two of the paper we provide background material on steganography, including measurements of the encoding rate as well as several techniques which have been used. Section three describes the reverse engineering process, with both static and dynamic analysis and their use for obfuscation and tamper proofing. Our methods are described in section four, and results and future work follow in succeeding sections.

2. BACKGROUND

Steganography using executable code as a hidden message has been an active research area for some time, and we are not the first researchers to propose executable programs as cover objects for steganographic purposes. Other researchers have explored program

stego, but mainly for hiding arbitrary data. For example, Shin et al. have hidden data in executable files by the simple method of adding additional sections to Windows Portable Executable (PE) files [20]. A similar method, but with encryption, is proposed by Zaidan [26,27,29]. The program Hydan uses semantically equivalent variants of instructions in order to hide data in executable code. El-Khalil and Kerymytis [8] report around a 1/110 bit encoding rate for Hydan, but show practical applications for the technique nonetheless. In more recent work, Anckaert et al. [30] describe a more systematic approach to Hydan and they identify three specific redundancies that can be exploited in the cover object (executable programs) to achieve more effective capacity and hiding. Their analysis along with that of Wright [31] point out the relative insecurity of Hydan from a detectability viewpoint and Anckaert's approach showed a 1/88 bit encoding rate on average that was architecture neutral.

We are also not the first researchers to propose using executable code as both the secret message and the cover object. Lu et al. [38] present RopSteg as a viable tool for embedding one program in another using return-oriented programming (ROP) [39]. Their approach utilizes typically nefarious techniques for finding code gadgets such as the Galileo algorithm to instead construct unintended ROP code segments. RopSteg illustrates a successful approach to executable steganography that are hidden from static analysis. However, they also acknowledge that current advances in defensive mechanisms to prevent ROP-based chaining might actually flag RopSteg as malicious code in the future, mainly because ROP has been seen as only useful for malicious programs. Ma et al. [35] further extend RopSteg and provide a concrete realization for using ROP execution as a watermarking scheme. Their work shows how ROP can be used advantageously as a dynamic watermarking approach because 1) there are no special data structures or added code need to support it; 2) there are no special code insertions that are independent from the original program; and 3) an external extractor does not need embedded information in the program to find the watermarked portion of code. Ultimately, their technique takes advantage of ROP gadgets which are built into the data region of the watermarked program. Our approach avoids issues with ROP detection as well as avoiding the need for data region changes.

Our research combines diverse fields of study, practical skill, and theory. First, steganography is concerned with how to conceal information in plain sight [28,32,33,34]. Reverse engineering and static/dynamic analysis [6,7,19,22,23], on the other hand, are practically learned skills that build on the theory of program analysis and compilers. Our aim is to perform steganography to conceal a program, and to prevent that program from easily being discovered in the reverse engineering process. Let us detail the reverse engineering process first, and include insights into the x86 instruction set in the process.

3. REVERSE ENGINEERING

Reverse engineering, also called back engineering or simply reversing, is the process of extracting knowledge and/or design information from a man-made artifact and reproducing something based on the extracted information [7]. Typical information recovered from reverse engineering may include knowledge about what tools were used to create the artifact, who the author of the artifact might be, or what design components were used in construction. In this proposal, we are concerned with a specific type of reversing: extracting knowledge about how a computer program operates, by looking at the underlying low-level machine code. Thus, reverse engineering in our context is the process of taking a binary executable artifact, a compiled program, and recreating a human-readable representation of the assembly code. Analysts use two primary techniques for reversing: 1) static analysis, where the program is examined without actually executing it, and 2) dynamic analysis, where the program is run under control of another, which monitors the behavior of the artifact.

Eilam [7] describes two common applications of reverse engineering in the software world: security-related and software development–related. From a security perspective, the technique can be used to assess resilience of applications from adversarial attacks like cracking and piracy; it can likewise be used to understand the working of malicious programs (malware). From a software development perspective, reversing can be used to evaluate software quality, develop competing software, or achieve interoperability with proprietary software. Both white-hat ethical hackers and black-hat hackers use reverse engineering as a foundational tool in program analysis [17]. In either case, engineers use tools to examine a program in an attempt to visualize the high-level code that was used to create it, and then use this knowledge to search for weaknesses or exploits, or to determine whether the code is malicious.

3.1 Static and Dynamic Analysis

Many tools exist to aid in the reverse engineering of software. The most popular include IdaPro [9], OllyDbg [18], udcli [24], objdump [15], as well as others. These tools are quite sophisticated, employing several methodologies to determine how the code operates. IdaPro is primarily a static analysis tool, although frequently used simultaneously with a debugger, while OllyDbg is mainly used for dynamic analysis. The udcli and objdump programs are simple command-line static disassembly tools. Static analysis is performed without actually executing the code and derives properties that hold true for all executions of a program. Disassembly, decompilation, control flow analysis, and data flow analysis are examples of static analysis—all of which produce conservative but potentially imprecise information [17]. Properties derived from dynamic analysis, on the other hand, only hold for the particular executions that are observed. This leaves a potentially large number of potential executable pathways within the code which will not be analyzed. Debugging, tracing, emulation, and profiling are examples of dynamic analysis—all of which produce non-conservative information that is always precise [17].

In static disassembly, the process starts at the program entry point by converting the binary (machine code) data back into assembly language, and then performing static analysis on the assembly to glean the control flow of the original program [6]. Disassemblers typically proceed in one of two ways: a linear scan method or a tree-based method [14]. Linear scan assumes that the instructions are contiguous and simply converts the entire instruction section back to assembly language one instruction at a time. The tree-based method uses intelligence to queue the targets (destinations) of jump instructions, starting the reverse assembly process at each target. Current tools such as IdaPro combine the two approaches. Others such as the Linux objdump utility perform a linear scan only. Some tools will also provide a block flow diagram similar to the Control Flow Graph (CFG) used by the compiler when the program was initially built. As long as the target of a jump instruction is known, the reversal process is relatively easy. However, jumps to addresses that are calculated within the program are not known through static analysis, and, as a result, some code may be missed. In the x86 architecture there are different types of jump instructions [1,10] which can be unconditional or conditional, a relative or absolute

address, and may be indirect. It is thus not possible to determine the target of every jump using only static analysis.

3.2 Obfuscation and Tamperproofing

Clever malware authors attempt to obfuscate, or hide, what the program actually does in order to thwart reverse engineering. Armored malware can utilize several mechanisms in order to determine at runtime whether the program is executing in a controlled environment, such as a virtual machine (VM) or under the control of a debugging program. Often, malware that is executed in a debugger can change program behavior to execute benign code so that an engineer trying to determine operational characteristics does not see what happens when the code is running normally. Malware can also use obfuscation or polymorphism to hide code by using packing programs such as UPX [25]. A packing program extracts the actual machine code from a hidden and compressed portion of the file only when the program is executed. Static analysis of a packed program will only show the unpacking code that expands the actual program; it is necessary to run at least some of the program in order to get a true picture of what the executable is capable of doing, and then to use dynamic analysis tools to trace through the program. In the general case, the privacy of executable programs can be enhanced in part by various techniques for performing software obfuscation. These techniques fall into three categories:

1. Obfuscation at the source code level, in order to hide variable names, strip comments, remove indentation that is indicative of program structure, and other source level modifications.
2. Obfuscation at an intermediate level, altering the way the program accomplishes certain tasks, but in such a way as to not modify the observable behavior of the program [5,13].
3. Finally, obfuscation at a binary level. In this low-level obfuscation, the assumption is that the reverse engineering process is concerned primarily with recreating an accurate assembly language representation of the binary code. Here, techniques include inserting conditional jumps where a tested predicate is known to be false, and inserting junk data into key areas in the binary so as to throw off disassemblers in architectures with varying length instructions.

Various methods for binary obfuscation have been examined previously: Nagra and Collberg [17] as well as Balakrishnan [4] provide extensive breadth and depth of such techniques.

In this research we are primarily concerned with obfuscating code at a binary level using steganography to embed instructions into a program. Traditional reversing tools such as IdaPro [19] are often used as benchmarks to evaluate correct disassembly when binary obfuscation is in view, and we use this tool to demonstrate our results below.

3.3 Algorithmic Foundations

We introduce our idea in more detail by first describing some obfuscation techniques unique to the x86 architecture and which are known. Techniques for fooling static analysis include those detailed by Aycock [2,3] and those outlined in the textbook by Sikorski [22]. These methods are designed to take advantage of limitations of the algorithms used in static analysis, including the inability to detect a jump target that is calculated at runtime. Linear disassembly algorithms that do not consider jump targets are particularly vulnerable by the addition of "junk" bytes used to throw off the disassembly process. Certain values are more effective [21] but the idea in these works is to fool the disassembler into mistaken identification of instructions. For example, a byte representing the first portion of an instruction that is N bytes in length will cause the next N-1 bytes to be assumed as part of the instruction. In fact, these may contain shorter instructions that are now missed by the reverse assembly process. An example of this technique is shown in Figure 1:

Memory	Instruction
31 ED	XOR EBP, EBP
49 89 D1	MOV R9, RDX
5E	POP RSI
48 89 E2	MOV RDX, RSP
05 31 ED 49 89	ADD EAX, 0x8949ED31
D1 5E 48	RCR dword [RSI+0x48],1
89 E2	MOV EDX, ESP

Figure 1. Addition of One Junk Byte

Note that the only difference between the first and second descriptions of memory is the addition of 0x05. Inserting this additional byte obscures four instructions; the reverse assembler incorrectly assumes that this is a three-instruction sequence as shown. After the last instruction the reverse assembler will again synchronize with the correct instructions in the program. Since inserting an additional byte will cause the disassembler to misrepresent the instruction stream, a logical question is what bytes to use in order to maximize the disruption. In previous research we created a unique version of the GCC compiler which automatically inserts "junk" between basic blocks [16]. We examined Linux utility binaries in x86-64 format using objdump to determine these optimal "junk" values, some of which are illustrated in Table 1. Here, "HI" refers to the average number of Hidden Instructions.

Table 1. Average Count of Hidden Instructions by Junk Byte

Value	Avg HI	Value	Avg HI	Value	Avg HI
0xa0	3.281	0x0F	2.444	0x35	2.342
0xa1	3.281	0x05	2.342	0x3d	2.342
0xa2	3.281	0x0D	2.342	0x68	2.342
0xa3	3.281	0x15	2.342	0xa9	2.342
0x69	2.911	0x1d	2.342	0xb8	2.342
0x81	2.911	0x25	2.342	0xb9	2.342
0xf7	2.561	0x2d	2.342	0xba	2.342

By examining this collection of Linux executable programs, we also observed that the average instruction length for x86-64 with Linux executable programs is 2.57 bytes per instruction. In general, x86-64 (and x86-32) is self-repairing from the reverse engineering standpoint. A byte of "junk" may cause a few instructions to be misrepresented, but only a few; after typically three instructions, occasionally five or six, the disassembly tends to resynchronize [12,14,23].

Another approach in hiding instructions is described by Jamthagen, et al. [11]. The authors describe a Main Execution Path (MEP), and a Hidden Execution Path (HEP), and break instructions down into three components, called XX, YY, and ZZ. The XX portion represents the part of the instruction including the x86 prefixes, the actual opcode, and addressing modes that cannot be changed. The portion YY is described as the portion of the instruction that can be changed, such as immediate operands or addresses which are included in the instruction. They note that YY needs to be large enough to hide the instructions in their HEP. Likewise, the ZZ portion of the instruction must be able to take any value. But

further, the combination of this ZZ plus the following XX must decode into another instruction.

Jamthagen also partitions the x86 instructions into select groups that might qualify for the main execution path and that meet these requirements: (1) the instructions must overlap each other and must never be aligned such that two instructions end at the same byte, and (2) both execution paths must contain valid instructions. It is necessary to select the last instructions in both the hidden and main execution paths carefully so that the final instructions in both leave with the next instruction in common. They further classify the x86 instructions into groups that may be used for the MEP. In part this is necessary due to the semantics of the instruction, a problem we also encounter below.

4. BINARY STEGANOGRAPHY

Given this foundational background, we now discuss our research effort into executable steganography. As definitions, we refer to different types of instructions by two names: Hiding Instructions (HI) are those that appear when reverse engineering the input program. These are the instructions in the program that a reversing tool such as IdaPro will show to the user. Steganography Instructions (SI) are those that are hidden.

Our method proceeds as follows:

1. First it is necessary to select the hiding instructions by writing high level code which preferably has a large number of 64-bit constants. The HI code is compiled to assembly language and assembled so that an assembly listing file can be created. The listing file is used to separate the operands from the operation codes in the instructions. Details on this step are in section 4.1.
2. Next, the selection of instructions eligible for SI is very limited, and as such the code for SI is coded in assembly language using this restricted set. The Si instructions are assembled so that we again obtain a listing file. This is described in section 4.2.
3. Each line if the listing file for HI is read, and each line of SI is also read. The bytes forming the opcodes in the listing of SI are placed into eligible operands in the instructions of HI, creating a new assembly file which is the combination of the two. This is described in section 4.3.
4. The resulting file is assembled and linked into the application as usual.

Our methodology relies on examining instructions in HI to determine where instructions in SI can be hidden. For example, if one wishes to hide the instructions shown in the top half of Table 1, a valid method is to use the junk byte of 0x05 as shown in the bottom half of Figure 1. This will cause a disassembler to assume that the first assembly language command, which is a hiding instruction, is the instruction `ADD EAX,0x8949ED31`. If the program were to jump to the beginning of the block, the 0x05 byte, the processor would in fact execute this, adding the constant to the indicated register. It would then execute the next instruction: `RCR dword [RSI+0x48],1`.

The selection of instructions is restricted by several factors so that if the main execution path in HI is executed it does not crash the program. As such these instructions must be selected with care. For example, loading a register from a memory location well outside the range of the program will look fine in a static disassembly, but will obviously cause a program fault if it is actually executed. Candidate instructions are, for example, those that might load something into a register but from a constant value as opposed to from a memory location. Thus, possible instructions include such operations as LEA (Load Effective Address), CMOVcc (Conditional Move) with a known false condition, SETcc (Conditional Set) with a known false condition, NOP (No Operation), and others. However, one can assume that the person wishing to hide SI is also the same person that is the author of HI. If we accept a slightly more "loose" view and assume that instructions are OK as long as they do not cause a program crash, then candidates also include instructions which may destroy register contents but have no disastrous side effects. An example is a MOV instruction with an immediate operand, or a conditional jump that is based on a known-false predicate.

With this idea in mind we now describe each step in turn.

4.1 Prepare Hiding Instructions

Based upon the knowledge we have of which candidate instructions are useful for HI, the user first creates a program either in assembly language or more likely in a language such as C. The program should be biased towards providing large amounts of the candidate instructions – those with operands which can be replaced with hidden instructions. An easy way to accomplish this is to include 64-bit values in calculations, and we show an example function in Figure 2:

```
unsigned long long hash( const char *key )
{
  unsigned long long mash;
  mash = strtoull( key, NULL, 10 );
  mash += 0x1111111111111111LL;
  mash |= 0x2222222222222222LL;
  mash -= 0x3333333333333333LL;
  mash &= 0x4444444444444444LL;
  mash ^= 0x5555555555555555LL;
  mash *= 0x6666666666666666LL;
  return( mash );
}
```

Figure 2. Example HI Program Code

The program is next compiled into assembly language and then assembled. We do not seek the actual object code for this step, but rather ask the assembler to create a "listing file". This file will contain not only the original mnemonics for the generated instructions but also the hex opcodes for the instruction, as in the Figure 3. (Only a few lines are shown and the figure is reformatted; some lines are split across two.)

Using a simple text manipulation program will yield a file with only the operation codes and the original assembly mnemonics included.

The next step is to identify candidate areas in the opcodes where SI can be inserted. These are mainly the operands in the instructions such that changing the value of the operand will have no ill effect on the program. In order to accomplish this we use a program based on a modified library called Udis86 [24]. This library allows one to provide a byte stream and retrieve (among other things) a string representation of the instruction corresponding to the bytes. The library was modified so that if we provide it with a set of bytes it will replace certain portions of the opcode to indicate the operands. Specifically, the original hex is returned to the caller but with certain characters replaced as in table 2.

```
55              push rbp
4889E5          mov rbp, rsp
4883EC20        sub rsp, 32
48897DE8
                mov QWORD PTR [rbp-24], rdi
488B45E8
                mov rax, QWORD PTR [rbp-24]
BA0A0000 00     mov edx, 10
BE000000 00     mov esi, 0
4889C7          mov rdi, rax
E8000000 00     call strtoull
488945F8
                mov QWORD PTR [rbp-8], rax
48B81111 11111111 1111
                movabs rax, 1229782938247303441
480145F8        add QWORD PTR [rbp-8], rax
48B82222 22222222 2222
                movabs rax, 2459565876494606882
480945F8        or QWORD PTR [rbp-8], rax
```

Figure 3. HI Assembly Listing

Table 2. Byte Indicators for Candidate SI Content

I	Position of a 1-byte immediate value
J	Position of a 2-byte immediate value
K	4-byte immediate value
L	8-byte immediate value
W	Position of a 1-byte relative offset
X	Position of a 2-byte relative offset
Y	4-byte relative offset
Z	8-byte relative offset (impossible)

The above partial listing in Figure 3 using this notation gives the results shown in figure 4. The operation codes are retained but the resulting strings indicate which operands, and what type and length, are in each operation:

```
55
4889E5
4883ECII
48897DE8
488B45E8
BAKKKKKKKK
BEKKKKKKKK
4889C7
E8KKKKKKKK
488945F8
48B8LLLLLLLLLLLLLLLL
480145F8
48B8LLLLLLLLLLLLLLLL
480945F8
```

Figure 4. HI Assembly Operand Marking

Although in theory the relative offsets present in the instruction are candidates for hiding SI, they suffer from the drawback that 1) they are usually too short, and b) represent addresses which are not resolved until the program is linked. Thus, the real candidates for hiding executable programs via steganography are the immediate constants, represented by Ks and Ls in the above.

4.2 Prepare Steganography Instructions

Next we go down a similar path for the instructions in SI. Here we will assume that the author of SI is writing the code directly in assembly language, since a key factor is to select instructions which are very short. Several tricks can be used to minimize the instruction sizes; a few examples are noted here.

- Exclusive or-ing a register with itself is preferable relative to loading zero into that register.
- Similar, to load one or two, etc. first zero the contents and then increment the register.
- To retrieve the current address, call the next instruction and then pop the return address from the stack. This can be used to access memory locations by offset from the current position.

One can consult the intel or AMD opcode map documents, or online versions such as at x86asm.net. But to aid the malware author – and who wants a stressed-out malware author? – we created a program using the Udis86 library that generates all single-byte, or two-byte, or … instructions in x86. The output from this program can also serve as a helpful reference.

The SI program is again assembled primarily so that we can obtain a listing file with the appropriate operation codes as shown in figure 5:

```
#          shell: .asciz "/bin/sh"
4831C0            xor   rax, rax
B03B              mov   al, 59
4831F6            xor   rsi, rsi
4831D2            xor   rdx, rdx
E800000000 here:  call  .+5
5F                pop   rdi
83C700            .byte 0x83, 0xc7,
                        (shell-here)-5
0F05              syscall
```

Figure 5. Example SI Program

In this example, from a 64-bit Linux machine, we will start a shell which runs at the privilege level of the user that executes the SI code. The parameters are passed in registers and most of them in this case can be null/zero. The string for "`/bin/sh`" is commented out here but will be manually added. Note that the GNU assembler is limited; although the constant displacement "`shell-here`" is small, and the labels are defined before they are referenced, this assembler assumes the displacement must be 32-bits. In the case of the Linux GNU assembler it is necessary to hand-code an add instruction which has an 8-bit displacement. On our Windows implementation the assembler will correctly gauge the short distance.

4.3 Instruction Weaving

Now that we have an appropriate listing file for HI and SI we can intermingle the instructions. Broadly speaking the following steps are employed:

1. Each line if the listing file for HI is read, and the following information is retained in a list of tuples which each contain:
   - The original opcode for the instruction.
   - The opcode with operands replaced per Table 2.
   - The original assembly language string, used if this instruction in HI is not a candidate for replacement.
   - Various "commands" stored as strings that are used to mark locations for later processing.
2. Each line of the listing file for SI is read, and the following information is retained in a list of tuples:
   - The number of opcode bytes for the instruction.
   - The original assembly language string.

3. Each instruction in HI is examined and if no opcode from SI can be hidden, the command for the tuple is set so that the instruction is not considered.
4. Otherwise,
   a. Replace the assembly instruction in HI with a declaration of bytes corresponding to the bytes in the operation code that will not be replaced.
   b. Insert a "target" command into the list of HI data.
   c. Select as many of the next instructions from SI as will fit in the operand for the HI instruction, leaving two bytes available for later use.
   d. Insert instructions from SI into the list of tuples for HI.
   e. Insert a "needjump" command into the list of HI.
   f. Insert as many additional bytes as are necessary to "fill" the original instruction length.

The process is repeated until either all instructions in SI have been inserted, or the end of the list of HI instructions is encountered. In the latter case this implies that SI is too long (or has instructions which are too long) for the number of instructions in HI, which is considered a fatal error.

The final step is to traverse the list of HI tuples in reverse order. At each "target" we note the byte offset in the machine code, and at each "needjump" we insert a short relative jump with the appropriate byte offset to go to the next hidden instruction.

To demonstrate the weaving process, examine the HI code in figure 3 above. We wish to embed in this function a hidden payload, shown in figure 5, which can be used to start a command prompt from within the program, if the entry point to the function is offset to the first instruction in SI. As described previously, we have compiled HI and assembled SI. The (partial) output of the method is shown in figure 6:

```
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
        mov     rax, QWORD PTR [rbp-24]
        mov     edx, 10
        mov     esi, 0
        mov     rdi, rax
        call    strtoull
        mov     QWORD PTR [rbp-8], rax
# (Was: movabs rax, 1229782938247303441)
        .byte 0x48, 0xB8
        xor     rax,  rax
        mov     al, 59
        .byte   0xEB, 7 # Go to next SI
        nop
        add     QWORD PTR [rbp-8], rax
# (Was: movabs rax, 2459565876494606882)
        .byte 0x48, 0xB8
        xor     rsi, rsi
        .byte   0xEB, 9 # Go to next SI
        nop
        nop
        nop
```

Figure 6. Example SI Program

Although there is no reason to fill any remaining operand space with something special, we elected to fill these areas with no-operation commands; any byte will actually do for these areas since they will not be executed within SI.

As a check, a partial disassembly for HI is shown in Figure 7.

Memory
48 89 45 F8 48 B8 48 31 C0 B0 3B EB 07 90 48 01 45 F8 48 B8 48 31 F6 48 31 D2 EB 06
Instructions

mov [rbp-8], rax
mov rax, 9007EB3BB0C03148h
add [rbp-8], rax
mov rax, 6EBD23148F63148h

Figure 7. Reverse Assembled Hiding Instructions

Execution continues in sequence after this. If these instructions do not cause the program any harm, then at runtime they could be executed, while also allowing us to execute the SI sequence starting with the instruction `XOR EBP,EBP`. What is necessary is to jump to the correct entry point for the sequence of instructions that is desired. But if the entry point is offset in memory, skipping the 0x48, 0x89, … then the hidden payload, SI, is executed instead as shown in Figure 8 starting at 0x48, 0x31, ...

Memory	Instructions
48 89 45 F8 48 B8	(Skipped)
48 31 C0	xor rax, rax
B0 3B	mov al, 3Bh
EB 07	jmp short +7
90 48 01 45 F8 48	(Skipped)
B8	xor rsi, rsi
48 31 F6	xor rdx, rdx
48 31 D2	jmp short +6
EB 06	

Figure 8. Reverse Assembled Hiding Instructions

We can see Si hidden in HI only by starting the disassembly process at the correct address. The ramification of instruction hiding is that if we choose the bytes with care, the disassembled HI can be made to look innocuous: more importantly, if we do select bytes carefully we can guarantee the program will not fail if it is executed.

5. RESULTS

We have implemented our code weaving using both C# on a Windows machine and also the tools available from a Linux installation. There are variations in the format of the listing files across these two platforms, but we are not addressing these issues here. Rather, the next question is whether the steganography actually is successful in hiding SI within HI. We turn to two tools, IdaPro version 6.9 [9] and the Linux objdump program [15] and in these figures show the Linux executable. Figure 9 demonstrates that IdaPro has correctly decoded the instructions in HI.

```
                public hash
hash:                                   ; CODE XREF
                push    rbp
                mov     rbp, rsp
                sub     rsp, 20h
                mov     [rbp-18h], rdi
                mov     rax, [rbp-18h]
                mov     edx, 0Ah
                mov     esi, 0
                mov     rdi, rax
                call    _strtoull
                mov     [rbp-8], rax
                mov     rax, 9007EB3BB0C03148h
                add     [rbp-8], rax
                mov     rax, 90909009EBF63148h
                or      [rbp-8], rax
```

Figure 9. IdaPro x86-64 Reverse Assembly

The objdump program output is shown in figure 10:

```
push   rbp
mov    rbp,rsp
sub    rsp,0x20
mov    QWORD PTR [rbp-0x18],rdi
mov    rax,QWORD PTR [rbp-0x18]
mov    edx,0xa
mov    esi,0x0
mov    rdi,rax
call   400590 <strtoull@plt>
mov    QWORD PTR [rbp-0x8],rax
movabs rax,0x9007eb3bb0c03148
add    QWORD PTR [rbp-0x8],rax
movabs rax,0x90909009ebf63148
```

Figure 10. objdump x86-64 Reverse Assembly

We also tested the executable with the open source edb debugger as well as the GNU debugger gdb; figure 11shows the results from edb; the disassembly from gdb is similar to figure 10.

```
imul ebp, dword [rsi+0x2f], 0x55006873
mov rbp, rsp
sub rsp, 0x20
mov [rbp-0x18], rdi
mov rax, [rbp-0x18]
mov edx, 0xa
mov esi, 0
mov rdi, rax
call demo!strtoull@plt
mov [rbp-8], rax
movabs rax, -0x6ff814c44f3fceb8
add [rbp-8], rax
movabs rax, -0x6f6f6ff61409ceb8
or [rbp-8], rax
```

Figure 11. Open Source edb Debugger Reverse Assembly

In order to provide an "acid test" the demonstration program that calls our "hash" function can call it via the normal entry point, which returns a 64-bit number based in part on the string passed in. Alternatively, the destination address for the function can be offset by the correct number of bytes and the instructions in SI are executed, resulting in a shell prompt.

6. NOTES and FUTURE WORK

Our technique works but suffers from shortcomings. Some of these shortcomings cannot be avoided, nor can they be dealt with. Others could be improved upon in future work. First we itemize several areas which are either potential improvements or notes about inherent shortcomings:

- The difficulty as we know is the requirement that instructions be sufficiently short that they can fit inside an operand. Since the majority of these immediate operands are 32-bit quantities, even in a 64-bit architecture, the selection of instructions is very limited. It is necessary for the author of the hidden code to craft the program with extreme care, by hand, in assembly language. While there is no special preparation needed for the code – it uses standard tools in order to be assembled – the author is extremely restricted in instruction selection.
- The number of available "slots" to place these instructions means that the one-byte relative offset in our jump may not be within range. Longer relative offsets would then eat into the bytes we need for the already restricted instruction set.
- The GNU assembler, "`as`", simply does not understand optimizing the size of jump displacements based upon where the jump target is located. It assumes in almost all cases that a 32-bit relative address is needed. It's thus necessary to craft our own jumps, a byte at a time, on Linux.
- Not all immediate constants are created equal. An example is the instruction "`mov edx, 10`", with an opcode `BA` and operand `0A000000`. Replacing this 32-bit immediate operand with something else obviously causes different results when calling "`strtoull`".
- Similarly, consider setting up the stack frame with "`sub rsp, 32`". Replacing the 32-bit constant here in the function prologue with instructions will cause considerable grief at the epilogue when the stack needs to be adjusted back.
- On Windows, the assembly listing file contains the character "R" to tell us that the operand portion of an instruction is subject to relocation when the linker builds the executable. This is very handy because we can simply ignore those operands that will need relocations. On Linux it will be necessary to examine the object file in greater detail to determine which immediate operands are subject to relocation. A greater investigation of the Udis86 library code may also be a means to solve this on Linux.
- In our above example it is necessary to manually insert the string "`/bin/sh`" since it is too long to be hidden in an operand and still include a jump.
- A system call to start a command prompt in Linux is fairly straightforward, as shown in our example. The Windows "`CreateProcess`" call requires ten parameters.

While we have shortcomings we also recognize some advantages as well. Instruction weaving is an interesting concept but one which may be limited in general usage because of the restrictions itemized above. But utilizing the more esoteric instructions available in x86-64 such as those using MMX, 3Dnow, or others, and using 80-bit floating point numbers, may be some way forward relative to these problems. The technique may be applicable to other complex instruction sets as well, but with the design of new CPUs being dominated by instructions with fixed lengths the applicability here may be small.

Finally, we also point out that none of the instructions in SI are encrypted in any way. A mitigation strategy is thus to utilize a brute force approach to reverse engineering the program, starting a disassembly process at every byte in the section. Short sequences of valid instructions, which possibly also end in an invalid decoding, are ignored, while long sequences of valid instructions can trigger an alarm for further investigation. An interesting future research project would be to come up with a method to obfuscate the obfuscated instructions.

7. ACKNOWLEDGEMENTS

This work is partially funded by National Science Foundation awards 1811560 and 1811578 in the NSF 17-576 Secure and Trustworthy Cyberspace (SaTC) program.

8. REFERENCES

[1] AMD, *AMD64 Architecture Programmer's Manual - Volume 3: General-Purpose and System Instructions*

[2] Aycock, John, Rennie deGraaf, and Jacobson Jr, Michael (2006). "Anti-disassembly using cryptographic hash functions". *Journal in Computer Virology*, 2(1):79–85, 2006.

[3] Aycock, John, Juan Manuel Gutierrez Cardenas, and Daniel Medeiro Nunes de Castro (2009). "Code obfuscation using pseudo-random number generators".

IEEE 15th International Conference on Computational Science and Engineering, 3:418–423, 2009.

[4] Balakrishnan, Arini, and Chloe Schulze (2005). "Code Obfuscation Literature Survey", at http://pages.cs.wisc.edu/~arinib/writeup.pdf

[5] Chen, Haibo, et. al (2009). "Control flow obfuscation with information flow tracking" *MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, ACM New York.

[6] Chess, Brian and Jacob West (2007). *Secure Programming with Static Analysis*. Pearson Education.

[7] Eilam, Eldad (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons.

[8] El-Khalil, Rakan, and Angelos D. Keromytis (2004). "Hydan: Hiding Information in Program Binaries", *Lecture Notes in Computer Science, vol. 3269, Information and Communications Security*, pp. 187-199, Springer, 2004.

[9] HexRays *IdaPro*, http://www.hex-rays.com/products/ida/index.shtml

[10] Intel (2006). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, 2006.

[11] Jamthagen, Christopher, Patrik Lantz, and Martin Hell (2013). "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries", *2013 Workshop on Anti-malware Testing Research (WATeR)*, 978-1-4799-2476-9, IEEE.

[12] Krishnamoorthy, Nithya, Saumya Debray, Keith Fligg (2009). "Static Detection of Disassembly Errors", *Proceedings of the 16th Working Conference on Reverse Engineering*, pp. 259-268, IEEE Computer Society Washington, DC, USA. doi: 10.1109/WCRE.2009.16

[13] László, T. and A. Kiss (2008). "Obfuscating C++ Programs via Control Flow Flattening", *Annales Universitatis Scientiarum de Rolando Eötvös Nominatae*—Sectio Computatorica, pp. 3-19, May 2008.

[14] Linn, Cullen, and Saumya Debray (2003). "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2003),* Washington, DC, USA, Oct. 27-30, 2003.

[15] LinuxDevCenter at http://www.linuxdevcenter.com/cmd/cmd.csp?path=o/objdump

[16] Mahoney, William (2015). "Modifications to GCC for Increased Software Privacy", *International Journal of Information and Computer Security*, Vol. 7, No. 2, 3, 4.

[17] Nagra, Jasvir and Christian Collberg (2010). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (Addison-Wesley Software Security Series). Pearson Education.

[18] OllyDbg, at http://www.ollydbg.de/

[19] Pearce, Walter (2008). *Reverse Engineering Code with IDA Pro*. Elsevier Science.

[20] Shin, DaeMin, Yeog Kim, KeunDuck Byun, and SangJin Lee (2008). "Data Hiding in Windows Executable Files", *Proceedings of the 6th Australian Digital Forensics Conference*, Edith Cowan University, Perth Western Australia, December 2008.

[21] Shinn, Sara, and William Mahoney (2011). "Optimal Values for Disrupting x86-64 Reverse Assemblers", *International Journal of Computer Science and Network Security*, Vol. 11, No. 11, 2011.

[22] Sikorski, Michael, and Andrew Honig (2012). *Practical Malware Analysis, the Hands-On Guide to Dissecting Malicious Software.* No Starch Press.

[23] Udupa, Sharath, Saumya K. Debray, Matias Madou (2005), "Deobfuscation – Reverse Engineering Obfuscated Code", *12th Working Conference on Reverse Engineering*, IEEE.

[24] Udis86, at http://udis86.sourceforge.net/

[25] UPX, at https://upx.github.io

[26] Zaidan, A.A., B.B.Zaidan, Fazidah Othman (2009). "New Technique of Hidden Data in PE-File with in Unused Area One", *International Journal of Computer and Electrical Engineering*, Vol. 1, No. 5 December, 2009, 1793-8163.

[27] Zaidan, A.A., B.B. Zaidan, Hamid.A.Jalab (2010). "A New System for Hiding Data within (Unused Area Two + Image Page) of Portable Executable File using Statistical Technique and Advance Encryption Standared" (sic), *International Journal of Computer Theory and Engineering*, Vol. 2, No. 2 April, 2010, 1793-8201.

[28] Simmons, Gustavas J. (1984). "The Prisoners' Problem and the Subliminal Channel", *Advances in Cryptology: Proceedings of Crypto '83*, pp. 51-67, Springer, 1984.

[29] Haveliya, Asmita (2012). "A New Approach for secret concealing in Executable File", *International Journal of Engineering Research and Applications (IJERA)*, Vol. 2, Issue 2, Mar-Apr 2012, pp.1672-1674.

[30] Anckaert, Bertrand, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere (2004). "Steganography for Executables and Code Transformation Signatures", *Proceedings of the 7th international conference on Information Security and Cryptology (ICISC'04),* pp. 425-439, Springer-Verlag Berlin, Heidelberg, doi: 10.1007/11496618_31

[31] Wright, Craig S. (2008). "Detecting Hydan: Statistical Methods for Classifying the Use of Hydan Based Stegonagraphy in Executable Files", *SANS Institute Reading Room.*

[32] Cole, Eric (2003). *Hiding in Plain Sight: Steganography and the Art of Covert Communication*, Wiley, ISBN: 978-0471444497.

[33] Wang, Huaiquang and Shuozhong Wang (2004). "Cyber warfare: steganography vs. steganalysis", *Comm. of the ACM*, Vol. 47, Issue 10, October 2004, pp. 76-82.

[34] Cazalas, Joshua, and Todd R. Andel, and J. Todd McDonald (2014), "Analysis and Categorical Application of LSB Steganalysis Techniques ", *The Journal of Information Warfare*, vol. 13, no. 3, July 2014.

[35] Ma, Haoyu, Kangjie Lu, Xinjie Ma, Haining Zhang, Chunfu Jia, Debin Gao (2015). "Software Watermarking using Return-Oriented Programming", *Proceedings of the 10th ACM Symposium on Information, Computer and*

Communications Security (ASIA CCS'15), pp. 369-380, April 14-17, 2015, Singapore. doi: 10.1145/2714576.2714582

[36] Gómez, Roberto and Gabriel Ramírez (2010). "Using Digital Images to spread Executable Code on Internet", *10th International Conference on Innovative Internet Community Services (I2CS), Jubilee Edition 2010*, June 3-5, 2010, Bangkok, Thailand.

[37] Ibrahim, Rosziati and Teoh Suk Kuan (2011). "Steganography Algorithm to Hide Secret Message inside an Image", Computer Technology and Application, vol. 2, pp. 102-108.

[38] Lu, Kangjie, Siyang Xiong, Debin Gao (2014). "RopSteg: Program Steganography with Return Oriented Programming", *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY '14)*, pp. 265-272, ACM New York, NY, USA. doi: 10.1145/2557547.2557572

[39] Shacham, H. (2007). "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 (CCS 2007)*, Alexandria, VA, USA, Oct. 29-Nov. 2, 2007.