Evaluating Security of Executable Steganography for Digital Software Watermarking

J. Alex Mullins
School of Computing
University of South Alabama
Mobile, AL

jam1201@jagmail.southalabama.edu

J. Todd McDonald
School of Computing
University of South Alabama
Mobile, AL

jtmcdonald@southalabama.edu

William R. Mahoney
School of Computing
University of Nebraska at Omaha
Omaha, NE

wmahoney@unomaha.edu

Todd R. Andel
School of Computing
University of South Alabama
Mobile, AL

tandel@southalabama.edu

ABSTRACT

Man-at-the-end (MATE) attacks against software programs are difficult to protect. Adversaries have complete access to the binary program and can run it under both static and dynamic analysis to find and break any software protection mechanisms put in place. Even though full-proof protection is not possible practically or theoretically, the goal of software protection should be to make it more difficult for an adversary to find program secrets by increasing either their monetary cost or time. Protection mechanisms must be easy to integrate into the software development lifecycle, or else they are of little to no use. In this paper, we evaluate the practical security of a watermarking technique known as Weaver, which is intended to support software watermarking based on a new transformation technique called executable steganography. Weaver allows hiding of identification marks directly into a program binary in a way that makes it difficult for an adversary to find and remove. We performed instruction frequency analysis on 106 programs from the GNU coreutils package to understand and define Weaver's limitations and strengths as a watermarking technique. Our evaluation revealed that the initial prototype version of Weaver suffers from limitations in terms of standard benchmarks for steganography evaluation, such as its stealth. We found that this initial prototype of Weaver relied heavily on one type of instruction that does not frequently occur in standard programs, namely the mov instruction with an 8-byte immediate operand. Our instruction frequency analysis revealed a negative impact due to Weaver's over-reliance on this mov instruction.

1. INTRODUCTION

Software piracy has been a steadfast problem in today's digital age. Software piracy is defined as "the unauthorized use such as possession, copying, distribution, buying and selling of software without the consent of the developer or the developing company" [10]. It is a heavy burden on U.S. based software companies trying to sell their products worldwide. The Commission on Theft of American Intellectual Property reported in 2017 that "IP theft is one of the most pressing issues of economic and national security facing our country." [4] The total estimated global monetary damages due to software piracy in 2015 amounted to $52.2 billion with a "low-end estimate for the cost to U.S. firms is $18 billion" [4].

Not only is software piracy causing monetary damages, but it is also a vector to spread malware to those people using pirated software. The BSA Software Alliance states that "it is increasingly clear that these malware infections are tightly linked to the using unlicensed software – the higher the rate of unlicensed software use, the higher the likelihood of debilitating malware infection." [5] There is a 29% chance of being infected with a piece of malware when a person installs unlicensed, pirated software [5]. Malware infections are a significant problem for businesses. It is estimated to cost them $359 billion worldwide to deal with malware infections of their computer systems [5]. In some parts of the world, software piracy is very prevalent. During his 2007 thesis research, Zhu reported that some countries have piracy rates as high as 92% [26].

Software developers rely on surreptitious software protection techniques to combat software piracy and protect their intellectual property. Five general techniques can be employed to protect a program from an adversary: code obfuscation, software watermarking, fingerprinting, birthmarking, and tamper-proofing [9]. Surreptitious software protection is a branch of computer security and cryptography [9].

Code obfuscation deals with the ability of an adversary to analyze a program's semantic meaning. The goal of code obfuscation is to transform the original code (source or binary) into a semantically equivalent form that is harder for an adversary to analyze or reverse engineer. Harder to analyze means increasing the cost to the adversary. That could mean monetary, time, or computational cost.

Tamper-proofing handles the case where an adversary has made changes to a program. The program can run self-verification routines to validate its integrity. These self-verification routines are usually implemented by hashing code sections and comparing the resulting checksum to the checksum computed at compilation. If the self-verification routine fails, then the program will quit or behave unexpectedly, rendering the software useless.

Watermarking, fingerprinting, and birthmarking are all considered "after-the-fact" techniques because they are used in the identification of stolen intellectual property rather than in preventing adversarial attacks. Watermarking embeds a mark inside of a program such that it can be extracted at a later time to prove program authorship. Watermarking is well-known outside of executable programs. For example, there are watermarks for images, videos, and audio. Figure 1 below is an example of an image watermark [24]. Fingerprinting is similar to watermarking. It embeds a mark inside a program to identify the valid user/owner whom the program author has given a license to use the software. Birthmarking is used to detect code similarity. All of these techniques are designed so that they can be used as evidence in a court of law where a program author brings a case of intellectual property theft against a defendant. Collberg discusses one such case where IBM used birthmarking to sue a competitor for stealing their PC-AT ROM code [9].

Figure 1. Example image watermark [24]

Software watermarking is one of the focus areas in this paper. Software watermarking was a popular topic in the research literature and took off in the late 1990s and early 2000s. Outside of the research literature, there is not much talk of software watermarking or software watermarking implementations used for protecting general purpose applications. If a company uses software watermarking, they most likely keep quiet about that fact to not draw the attention of adversaries. Software watermarking intends to embed a secret mark into a piece of software and identifies the program author. However, a software watermarking algorithm needs an extraction component to be truly useful. If a watermark cannot be extracted, then it provides no use in a court setting where a program author is trying to prove his case to a judge. Zhu and Thomborson formalized the concepts of embedding and extraction, which are shown in Table 1 [25].

For any of the software protection techniques discussed above to be used by software developers, they need to be implemented in tools that are easy to use. Most of the obfuscation, tamper-proofing, and watermarking solutions discussed in the research literature lack usability. If a tool is not easy to use and does not integrate into existing build systems, then there is no chance that a developer will touch it. While there exist software protection solutions in the form of usable tools, they are often restricted to specific programming languages, limiting their usability.

Table 1. Watermarking concepts [25]

Concept	Definition
Embedding	Inserting a watermark into a program
Extraction	Successful extraction of the inserted watermark
Blind extractability	Extraction requires no additional information
Informed extractability	Extraction requires extra information such as the original unwatermarked program

Tigress is a tool initially built to test the idea of code diversity as a method of overwhelming an adversary in a Remote Man-at-the-end (R-MATE) attack [7]. It has also been used to test the ability of obfuscated programs to reduce the effectiveness of symbolic execution analysis by causing path explosion, path divergence, and complex constraints [2]. Tigress is a C diversifier and obfuscator that applies obfuscating transformation at the source level. It takes in a C source file and applies several different transformations to it and then outputs an obfuscated C source file. Tigress has an impressive number of transformations. These include code virtualization, code jitting, dynamic code jitting, control flow flattening, function merging, function splitting, function argument reordering, opaque branching, integer, and string literal encoding, integer data encoding, integer arithmetic encoding, and more. These obfuscation transformations could slow down an adversary. However, the most significant disadvantage to Tigress is that it is dependent on the C source file to apply the obfuscation transformation successfully.

Sandmark is “a tool developed to aid in the study of software-based software protection techniques.” [6] It provides many software protection techniques that have been talked about previously. It provides static program analysis, software watermarking, tamper-proofing, and code obfuscation. However, these protections are only available for Java bytecode.

Obfuscator-LLVM, as its name implies, is an obfuscation tool with “a set of obfuscating transformations implemented as middle-end passes in the LLVM compilation suite.” [16] This means that any language that targets LLVM IR can use the code transformations provided by Obfuscator-LLVM. This list of front-end languages includes Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift. Also, Obfuscator-LLVM can be used with any backend that LLVM supports. That architecture list includes x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, ARM-64, Sparc, Alpha, and MIPS [16]. The list of obfuscating transformations that Obfuscator-LLVM provides is instruction substitution, bogus control-flow insertion, basic block splitting, procedure merging, and code tamper-proofing [16]. Because the transformations target the LLVM IR level, many languages and architectures are supported and can be used by a large audience. An early version of it made a debut being used to obfuscate the evasi0n7 jailbreak for iOS devices.

Weaver is a tool built to provide software watermarking protection for programs. It is a collaborative effort between faculty and students at the University of Nebraska at Omaha and the University of South Alabama. It is in a similar vein as Obfuscator-LLVM and targets the LLVM IR level to apply the watermark and

steganography transformations. The wide array of languages and architectures available to LLVM is very appealing for a general-purpose software protection tool. Weaver has undergone an update from the first version to a new version, which changes the steganographic technique used to hide the watermark. The first version used an executable steganography technique termed instruction weaving [18]. The second version of Weaver uses basic block insertion and opaque branches to hide the watermark.

In this paper, we intend to show that version one of Weaver was insufficient for hiding a watermark because of the lack of stealth provided by its instruction weaving technique. The lack of stealth would allow an adversary to spot the 'hidden' watermark easily. This paper is organized as follows: Section 2 gives background on software protection techniques and Weaver; Section 3 will detail the approach we took to analyze the Weaver's stealth; Section 4 details our results; Section 5 gives our conclusions; Section 6 provides avenues for future work.

2. BACKGROUND

The techniques discussed previously, such as watermarking, steganography, and obfuscation, have been thoroughly explored in the research literature. These techniques are implemented in software to protect secrets in program code by slowing down an adversary from reverse engineering important algorithms or data structures which the program author considers as their intellectual property. It is important to note that these techniques cannot prevent an adversary from eventually recovering these secrets. Collberg et al. said, "there are no known algorithms that provide complete security for an indefinite amount of time. At present, the best we can hope for is to be able to extend the time it takes a hacker to crack our schemes." [9]

2.1 Software Watermarking

Software watermarking is an anti-piracy technique that helps a program author prove authorship of code obtained through illegitimate channels. During a court proceeding, the program author uses an extraction tool that has specific knowledge of the embedded watermark. This tool pulls out the watermark, which is presented to the court as evidence of program authorship.

Many of the first software watermarking techniques were published in patents filed by Davidson and Myhrvold [11], Moskowitz and Cooperman [21], and Samson [22]. Afterward, there was a surge of research work done in the late 1990s and early 2000s, focusing on software watermarking.

Software watermarking embeds a secret watermark M into a program P giving us P_W. Collberg and Thomborson define watermarking as [8]:

$$embed(P, W, key) = P_W$$

$$extract(P_W, key) = W$$

Hamilton and Danicic give five properties that they consider a "good" software watermarking system should have, which are outlined in Table 2 [14].

Table 2. Properties of Good Watermarking

Metric	Condition
Program Size	Program size must not be increased significantly
Program Efficiency	Program runtime must not be decreased significantly
Resiliency	Resistant to semantic preserving transformations
Invisibility	Sufficiently well-hidden to avoid removal
Extraction	Easy to extract by author

Zhu discussed the taxonomy of watermarking in his thesis work and classified them along many axes, and they are presented in Table 3 [26].

Table 3. Watermarking Taxonomies

Classification	Subclass
Purpose	• Prevention • Assertion • Permission • Affirmation
Extraction Technique	• Static • Dynamic
Fragility	• Robust • Fragile
Visibility	• Visible • Invisible
Detection Method	• Blind • Informed
Tamperproof-ness	• Tamperproof • Not Tamperproof

Classification by extraction technique (static vs. dynamic) is how many papers in the research literature differentiate software watermarking algorithms. Static watermarking schemes place a watermark directly in the code of an executable that can be extracted without running the program. An example of a static watermark would be the Monden et al. method of inserting a watermark in the Java bytecode of a program [20]. On the other hand, dynamic watermarking schemes place a watermark in the runtime execution state of the program, and it can only be extracted during the program's runtime. The most well-known dynamic watermarking scheme is the CT algorithm developed by Collberg, Thomborson, et al. [8]. The high-level algorithm flow shown in Figure 2 is from their paper "Dynamic graph-based software watermarking."

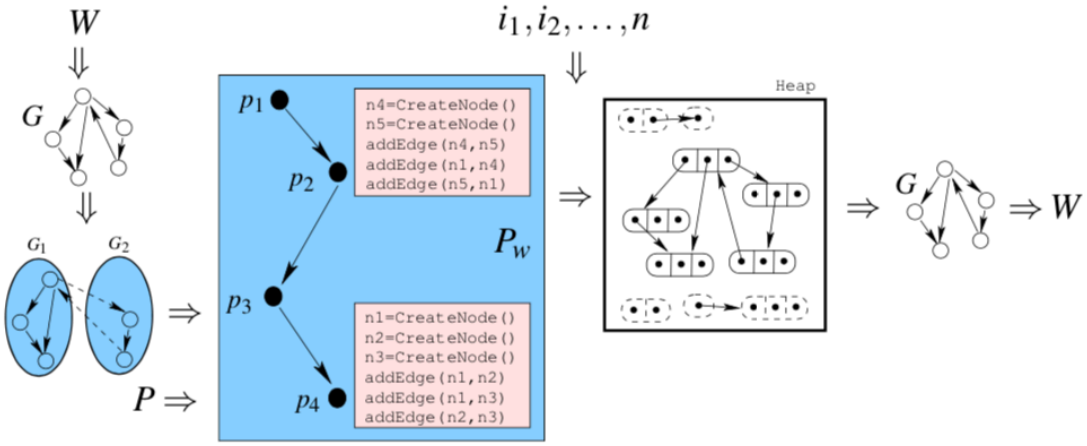

Figure 2. CT Algorithm of dynamic watermark built-in memory

From Figure 2, a graph is built beforehand that represents a unique watermark. This graph is transformed into code which can build the chosen graph. This code is then embedded into program *P*. Once the code is run, the graph will be built in the program's heap memory. The graph in memory is considered the actual software watermark.

Collberg and Thomborson cover three types of attacks against software watermarking: subtractive, distortive, and additive [8]. These attacks are performed by an adversary looking to remove a watermark from a watermarked program. In a subtractive attack, a watermark is wholly removed from the program. In a distortive attack, the program's code is semantically transformed to the point that the watermark cannot be extracted from the program. Additive attacks involve an adversary embedding their own watermark into an already watermarked program. An additive attack reduces the original author's claim in court because now an adversary can extract his watermark too.

2.2 Executable Steganography

Steganography is the age-old problem of trying to hide a secret message in a cover message. Steganography has a slightly different goal than that of cryptography as "… classical cryptography is about concealing the content of messages, steganography is about concealing their existence" [1]. Steganography is modeled by the Prisoners' Problem that was first discussed in this context by Simmons [23]. The problem is described as two inmates, Alice and Bob, who both want to plan a secret prison break, but they can only communicate through written messages that must pass through the prison's warden, Wendy. So, Alice and Bob must resort to embedding their secret plans in cover messages to not raise suspicion from Wendy. Wendy can read each message passed between the two inmates and can either allow the message to pass through or trash the message if she suspects anything nefarious.

Collberg and Thomborson give the strength of a steganographic algorithm by measuring its data rate, stealth, and resilience [8]. They go on to say, "data rate expresses the number of bits of hidden data that can be embedded within each kilobyte of cover message, the stealth expresses how imperceptible the embedded data is to an observer, and the resilience expresses the hidden message's degree of immunity to attack by an adversary" [8]. They go on to say that all of these metrics are trade-offs between each other.

There have been a few examples of hiding hidden messages in executables. The most well-known is Hydan created by El-Khalil and Keromytis [12]. It can embed a hidden message in "functionally-equivalent instructions" of a program's binary code and can be used to hide a static watermark, fingerprint, or any other secret message [12]. The downside of Hydan is its low data rate. The scheme can embed data at a 1/110 bit encoding rate (1 bit of hidden message data for 110 bits of cover message data) [12].

Executable steganography hides secret executable code inside a cover program's regular executable code. Figure 3 below is from the RopSteg paper and shows an excellent example of executable steganography [17].

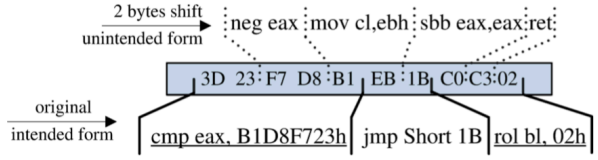

Figure 3. Executable steganography, hiding instructions

As can be seen, if instruction decoding starts at the byte 0x3D, then it would produce an instruction sequence of a cmp, jmp, and rol. However, if instead, instruction decoding starts at byte 0xF7, then a different instruction sequence is produced consisting of a neg, sbb, and a ret. This is what executable steganography is all about – hiding a different instruction sequence within a cover instruction sequence.

A novel executable steganographic technique discussed by Lu, Xiong and Gao uses return-oriented programming (ROP) to produce hidden code within a cover program [17]. They named their technique RopSteg.

2.3 Software Obfuscation

Software obfuscation is a software protection technique that transforms a program *P* into a semantically equivalent program *P'*. Effective obfuscation means the resulting program *P'* should be much harder to understand and analyze than the original program *P* [26]. Collberg et al. point out that obfuscation is a double-edged sword because it can be used by good guys to protect their IP, but can also be used by malware writers to hide their program's

malicious intent [9]. Zhu lays out four types of obfuscation techniques, which are detailed in Table 4 [26].

Table 4. Obfuscation Types

Obfuscation Type	Description
Design	Class merging, class splitting, and type hiding
Data	Split variable, merge variable, flatten array, fold array, etc.
Control-flow	Flattening, block reordering, method inlining, etc.
Layout	Lexical formatting, renaming, etc.

Obfuscator-LLVM is a well-known obfuscating compiler that offers obfuscation transformation for program protection [16]. Another implementation of an obfuscating compiler is from Mahoney, who used the GCC compiler to add obfuscation transformations [19]. Specifically, Mahoney's obfuscating GCC compiler provides three obfuscation techniques: jump hiding, block shuffling, and junk insertion [19]. He is also one of the authors of Weaver.

2.4 Weaver – Version 1

Weaver is an implementation of executable steganography developed by faculty and research students at the University of Nebraska at Omaha and the University of South Alabama. It has gone through a rewrite, but the first version is described in the paper "Leave it to Weaver" [18]. For the rest of this paper, we will refer to the first version of Weaver as (WV1) and the second version as (WV2).

WV1's executable steganography technique can hide secret code by exploiting the variable-length instructions of the Intel x86-64 instruction set architecture (ISA). ISAs are generally divided into two broad categories: complex instruction set computer (CISC) and reduced instruction set computer (RISC). A RISC architecture usually has fixed width instructions. For example, ARM ISA is a RISC architecture and has an instruction width of 4 bytes (in non-Thumb mode). The fixed width means that the starting address of every instruction will occur on a 4-byte boundary, making instruction decoding very simple.

On the other hand, Intel x86-64 is a CISC architecture and has variable width instructions that can be anywhere from 1 byte to 15 bytes [13]. The authors of Weaver conducted an empirical study and found that the Intel x86-64 instruction length of Linux executables averaged 2.57 bytes [18]. This variable width instruction encoding means that a valid instruction can occur at any byte boundary and can produce complications when disassembling [15]. WV1 takes advantage of this property to hide secret instruction sequences inside of cover instruction sequences.

Specifically, WV1 tries to hide the secret instructions in the bytes corresponding to large immediate operands of particular instructions in the cover program [18]. This hidden code can be jumped to so that it starts executing instead of the original code in the cover program. Without variable-width instructions, it would not be possible to jump in the middle of an instruction to the location of the hidden instruction. A high-level view of WV1 can be seen in Figure 4.

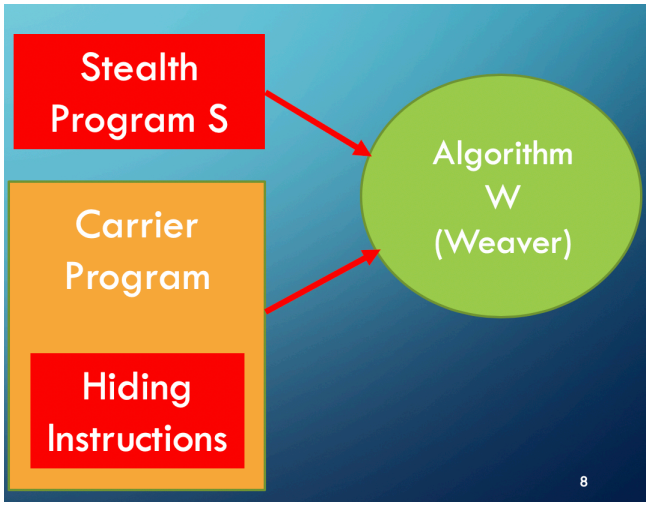

Figure 4. The high-level view of WV1

Weaver takes a specially crafted carrier (cover) program *C* that contains candidate hiding instructions *HI* that will eventually have their large immediate operands replaced with the instruction bytes from the stealth program *S,* which contains the stealth instructions *SI*. This carrier program is specifically made to contain candidate *HI,* and it is not part of the original program binary. WV1 only considers instructions from *C* with constant operands, and it classifies these instructions into two broad categories: immediate and relative offset operands. It further divides these instructions by the operand byte size. Table 5 shows the breakdown of these classifications [18].

Table 5. Classification for HI

Category	Byte Size	Operand Type
I	1	Immediate
J	2	Immediate
K	4	Immediate
L	8	Immediate
W	1	Relative offset
X	2	Relative offset
Y	4	Relative offset
Z	8	Relative offset

WV1, in theory, could use the relative offset instructions. However, it does not because "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." [18] And in fact, WV1 limits itself further due to the average Intel x86-64 instruction length of 2.57 bytes leaving just K and L category instructions that are useable as candidate *HI* [18].

Once WV1 has identified all of the candidate *HI* in *C*, it will replace the operand bytes of the *HI* with the binary code of *SI*. One requirement of WV1 is that it must insert a relative jump instruction as the last two bytes of each embedded sequence of *SI* to string together all the *SI* in *C*. Figure 5 shows an example of WV1 hiding code in the 8-byte immediate operands of the mov instructions. The

candidate *HI* is placed inside of a dummy 'hash' function (the carrier program) that is linked into the program's binary at compile time. This dummy 'hash' function does not provide anything useful to the binary other than to provide *HI* to hide *SI*. As can be seen, the 2-byte jump is necessary to avoid executing the add, or, and xor instructions that follow the immediate operands. Because of the required 2-byte jump, there are effectively 6 bytes to hide *SI* in *HI*.

```
hash:
00400796  55                push    rbp {__saved_rbp}
00400797  4889e5            mov     rbp, rsp {__saved_rbp}
0040079a  4883ec20          sub     rsp, 0x20
0040079e  48897de8          mov     qword [rbp-0x18 {var_20}], rdi
004007a2  488b45e8          mov     rax, qword [rbp-0x18 {var_20}]
004007a6  ba0a000000        mov     edx, 0xa
004007ab  be00000000        mov     esi, 0x0
004007b0  4889c7            mov     rdi, rax
004007b3  e8d8fdffff        call    strtoull
004007b8  488945f8          mov     qword [rbp-0x8 {var_10}], rax
004007bc  48b84831c0b03beb… mov     rax, 0x9007eb3bb0c03148
004007c6  480145f8          add     qword [rbp-0x8 {var_10_1} {var_10}], rax
004007ca  48b84831f6eb0990… mov     rax, 0x90909009ebf63148
004007d4  480945f8          or      qword [rbp-0x8 {var_10_2} {var_10_1}], rax
004007d8  48b84831d2eb0990… mov     rax, 0x90909009ebd23148
004007e2  480145f8          add     qword [rbp-0x8 {var_10_3} {var_10_2}], rax
004007e6  48b8e800000000eb… mov     rax, 0x9007eb00000000e8
004007f0  482145f8          and     qword [rbp-0x8 {var_10_4} {var_10_3}], rax
004007f4  48b85f83c7a1eb0c… mov     rax, 0x90900ceba1c7835f
004007fe  483145f8          xor     qword [rbp-0x8 {var_10_5} {var_10_4}], rax
00400802  488b55f8          mov     rdx, qword [rbp-0x8 {var_10_5}]
00400806  48b80f05eb899090… mov     rax, 0x9090909089eb050f
00400810  480fafc2          imul    rax, rdx
00400814  488945f8          mov     qword [rbp-0x8 {var_10_6}], rax
00400818  488b45f8          mov     rax, qword [rbp-0x8 {var_10_6}]
0040081c  c9                leave    {__saved_rbp}
0040081d  c3                retn     {__return_addr}
```

Figure 5. *SI* hidden in the *HI* of *C*

From Figure 5, when instruction decoding starts at 0x4007bc, a mov instruction is found:

mov rax, 0x9007eb3bb0c03148

However, when instruction decoding start 2 bytes further at 0x4007be (the start location of the 8-byte immediate operand) a new instruction sequence is found (the hidden *SI*):

xor rax, rax

mov al, 0x3b

jmp 0x4007cc

The jump to 0x4007cc puts the instruction pointer at the start of the next *SI* sequence in the middle of the mov instruction at address 0x4007ba.

2.5 Limitations

WV1 has a few limitations. First, the executable steganography technique is highly specific to the Intel x86-64 ISA. This keeps it from being employed on Android and iOS applications in which both run on the ARM architecture. One of the goals of Weaver is to be used with a wide variety of languages on various architectures. So this reliance on Intel x86-64 variable-width instructions makes that goal unrealistic.

Second, WV1 has a small pool of possible candidate *HI* from the K and L categories. The example code above in Figure 5 only uses the mov instruction with an 8-byte immediate operand (category L) that provides a total of 6 bytes for embedding *SI* and 2 bytes for the relative jump. WV1 can use candidate *HI* with 4-byte immediate operands (category K), but that leaves only 2 bytes for embedding *SI* with the last two bytes going to the required relative jump. As noted, there is an average instruction length of 2.57 bytes making the category K *HI* not usable in most situations. This means that in practice, WV1 only uses the category L *HI* to embed *SI*. The "Intel 64 and IA-32 Architectures Software Developer's Manual" only specifies one instruction that can take an 8-byte immediate operand, and that is the mov instruction [13]. This is a pretty impactful limitation because it potentially weakens the stealth of the executable steganography because there would need to be many of these 8-byte mov instructions to hide the *SI*. If 8-byte immediate operand mov instructions do not frequently occur in standard program code, then this might stick out to an adversary.

Both of these limitations of WV1 make it an unappealing solution. A move to rewrite Weaver to version 2 is underway. However, there was never an empirical test to see if the second limitation of WV1 possibly made the executable steganography stick out to an adversary. As Collberg and Thomborson noted, the stealth of a steganography system measures how imperceptible the hidden secret is from an adversary [8]. That is what this paper will look at in the next few sections. Do mov instructions with 8-byte immediate operands occur often enough in standard applications so that the *HI* does not seem out of place?

3. APPROACH

To test whether many 8-byte mov instructions will stick out to an adversary, we will perform an instruction frequency analysis on 106 binaries from the GNU coreutils (version 8.31) to see how often 1, 2, 4, and 8-byte immediate mov instructions occur. The GNU GCC compiler was used with an optimization level of 2. Bilar conducted a similar opcode frequency analysis to try to detect malware [3].

We will gather statistics on the 1, 2, and 4-byte immediate mov instructions to get a relative comparison to how often the 8-byte immediate mov instruction occurs. We assume that we have 600 bytes of *SI* that need to be hidden inside of the mov instruction's 8-byte immediate operands. These 600 bytes of instructions would be the graph code that would build a watermark in a program's heap memory. We arrived at 600 bytes by building a simple doubly linked list example in C with 32 nodes inserted (this closely resembles a graph-like structure). Each 8-byte immediate operand mov instruction will have effectively 6 bytes to hide *SI*. This means that at least 100 8-byte immediate operand mov instructions will be required to embed *SI* fully.

So, for instance, if standard applications only contain five 8-byte immediate operand mov instruction, then these 100 8-byte immediate operand mov instructions will be very noticeable.

The Binary Ninja disassembler will be used to gather the disassembly for each binary, and a Python script will be written to extract out the instruction frequencies from those binaries. If a mov instruction occurs, we first test whether it is moving an immediate value into a register or memory location and if so, we then classify that mov instruction based on a 1, 2, 4, or 8-byte immediate value. The Capstone disassembler tool will be used to classify each mov instruction.

4. RESULTS

After running the instruction frequency analysis, we gathered some interesting data. There was a total of 757,678 instructions contained in all 106 binaries. That number was spread over 138 unique instructions. The top ten most frequently encountered instructions are shown in Table 6.

Of the total 757,678 instructions, 253,521 (33.46%) of them were mov instructions. Of the 253,521 mov instructions, only 59,159 (23.33%) of them had immediate operands. The break-down of those mov instructions is shown in Table 7.

We categorize a mov instruction as one that does not have an immediate operand or has a 1, 2, 4, or 8-byte immediate operand.

The mov instruction with an 8-byte immediate operand occurred 1,928 times throughout the total 757,678 instructions or just 0.25% of the time.

Table 6. Top 10 Instructions

Instruction	Count	Percentage
mov	253521	33.46%
call	47421	6.26%
cmp	47056	6.21%
xor	44188	5.83%
lea	40750	5.38%
jmp	40668	5.37%
test	34151	4.50%
je	33097	4.37%
pop	27825	3.67%
add	26568	3.51%

Table 7. Break-down of MOV instructions and percentage of total instructions

Type	Count	Percentage
1-byte	12138	1.61%
2-byte	13	0.002%
4-byte	45080	5.95%
8-byte	1928	0.25%
No Imm	194362	24.65%

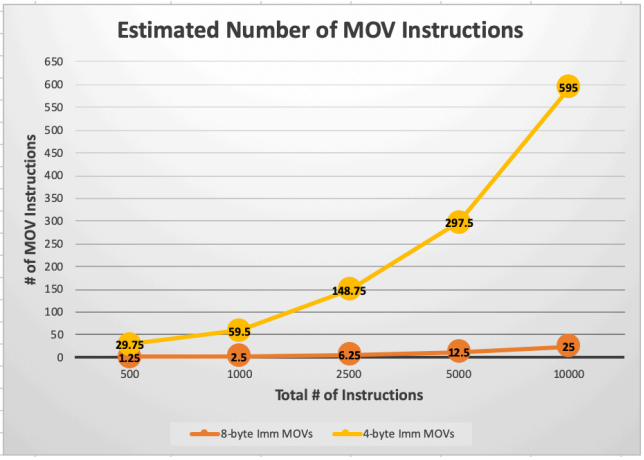

Figure 6. Estimated Number of MOV Instructions in Programs with Varying Number of Total Instructions

Figure 6 shows a graph of the estimated number of mov instructions with 4 or 8-byte immediate operands based on their respective computed averages in a program with varying numbers of total instructions of 500, 1000, 2500, 5000, and 10,000.

Based on the computed averages, a program with a total of 10,000 instructions would have roughly 25 8-byte immediate operand mov instructions and roughly 595 4-byte immediate operand mov instructions.

5. CONCLUSIONS

Based on the collected results, it seems that WV1 lacks the instruction diversity to stay hidden due to only using the 8-byte immediate operand mov instruction. To hide the 32-node watermarking code (600 bytes) would take around 100 8-byte immediate mov instructions as candidate *HI*. This means that to be effectively hidden (based on the computed average of 0.25%), a program would need to have around 40,000 instructions. Furthermore, this just takes into account the 8-byte immediate operand mov instructions added to the program's binary code to hide the watermarking code. It is expected that a program of that size would have additional mov instructions with 8-byte immediate operands that normally occur in the program's code. If WV1 were to use 4-byte immediate operand mov instructions to hide the watermarking code, it would need to use 300 mov instructions. However, it is not feasible to use 4-byte immediate operand mov instructions because of the average Intel x86-64 instruction length of 2.57 bytes. Most instructions would not fit in the left-over two bytes of the immediate operand after the 2-byte jump is inserted [18]. This conclusion shows that the Weaver author's suspicions were right that there is not enough stealth in the instruction weaving technique to stay hidden from a determined adversary.

6. FUTURE WORK

There is future work to be done in analyzing WV2 to test its security features. Especially looking at its ability to stay hidden from an adversary. WV2 is still being developed, and analysis work will start soon. Early prototypes of WV2 use LLVM/IR more heavily in its embedding process. The executable steganography technique being used is a basic block insertion algorithm that is hidden behind opaque predicates. Future work includes testing if these opaque predicates can be identified to narrow down the possible basic blocks of the hidden watermarking code.

7. ACKNOWLEDGMENTS

This research is supported by the National Science Foundation under the Secure and Trusted Computing (SaTC) grant CNS-1811578 and CyberCorps Scholarship for Service grant DGE-1564518. The project is a collaborative effort between the University of Nebraska at Omaha (UNO) and the University of South Alabama (USA).

8. REFERENCES

[1] Anderson, R.J. and Petitcolas, F.A. 1998. On the limits of steganography. *IEEE Journal on selected areas in communications*. 16, 4 (1998), 474–481.

[2] Banescu, S. et al. 2016. Code Obfuscation Against Symbolic Execution Attacks. *Proceedings of the 32Nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), 189–200.

[3] Bilar, D. 2007. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*. 1, 2 (2007), 156–168.

[4] Blair, D.C. et al. 2017. *Update To The IP Commission Report - The Theft of American Intellectual Property: Reassessments of the Challenge and United States Policy*. The National Bureau of Asian Research.

[5] Business Software Alliance 2018. *Software Management: Security Imperative, Business Opportunity*. Business Software Alliance.

[6] Collberg, C. et al. 2007. An empirical study of Java bytecode programs. *Software: Practice and Experience*. 37, 6 (2007), 581–641.

[7] Collberg, C. et al. 2012. Distributed Application Tamper Detection via Continuous Software Updates. *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), 319–328.

[8] Collberg, C. et al. 2004. Dynamic graph-based software watermarking. *TR04-08, Department of Computer Science*. (2004).

[9] Collberg, C. and Nagra, J. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

[10] Das, M. 2019. *Software Piracy*. Metropolia University of Applied Sciences.

[11] Davidson, R.I. and Myhrvold, N. 1996. Method and system for generating and auditing a signature for a computer program. US5559884A. Sep. 1996.

[12] El-Khalil, R. and Keromytis, A.D. 2004. Hydan: Hiding information in program binaries. *International Conference on Information and Communications Security* (2004), 187–199.

[13] (first) 2018. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel.

[14] Hamilton, J. and Danicic, S. 2010. An evaluation of static java bytecode watermarking. *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10), The World Congress on Engineering and Computer Science (WCECS'10), San Francisco* (2010).

[15] Hanov, S. 2009. Static analysis of binary executables. *University of Waterloo*. (2009).

[16] Junod, P. et al. 2015. Obfuscator-LLVM – Software Protection for the Masses. *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015* (2015), 3–9.

[17] Lu, K. et al. 2014. Ropsteg: program steganography with return oriented programming. *Proceedings of the 4th ACM conference on Data and application security and privacy* (2014), 265–272.

[18] Mahoney, W. et al. 2018. Leave It to Weaver. *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop* (2018), 6.

[19] Mahoney, W.R. 2015. Modifications to GCC for increased software privacy. *International Journal of Information and Computer Security*. 7, 2–4 (2015), 160–176.

[20] Monden, A. et al. 2000. A practical method for watermarking java programs. *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000* (2000), 191–197.

[21] Moskowitz, S.A. and Cooperman, M. 1998. Method for stega-cipher protection of computer code. US5745569A. Apr. 1998.

[22] Samson, P.R. 1994. Apparatus and method for serializing and validating copies of computer software. US5287408A. Feb. 1994.

[23] Simmons, G.J. 1984. The prisoners' problem and the subliminal channel. *Advances in Cryptology* (1984), 51–67.

[24] Watermark Software 2019. *Watermark Tiling for Strong Protection*.

[25] Zhu, W. and Thomborson, C. 2006. Extraction in software watermarking. *Proceedings of the 8th workshop on Multimedia and security* (2006), 175–181.

[26] Zhu, W.F. 2007. *Concepts and techniques in software watermarking and obfuscation*. The University of Auckland New Zealand.