Achieving Obfuscation Through Self-Modifying Code: A Theoretical Model

Heidi Angelina Waddell

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2020

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial fulfillment of the requirements for graduation from the Honors Program of Liberty University.

____
Melesa Poole, Ph.D.
Thesis Chair

____
Robert Tucker, Ph.D.
Committee Member

____
James H. Nutter, D.A.
Honors Director

____
Date

Abstract

With the extreme amount of data and software available on networks, the protection of online information is one of the most important tasks of this technological age. There is no such thing as safe computing, and it is inevitable that security breaches will occur. Thus, security professionals and practices focus on two areas: *security*, preventing a breach from occurring, and *resiliency*, minimizing the damages once a breach has occurred. One of the most important practices for adding resiliency to source code is through *obfuscation*, a method of re-writing the code to a form that is virtually unreadable. This makes the code incredibly hard to decipher by attackers, protecting intellectual property and reducing the amount of information gained by the malicious actor. Achieving obfuscation through the use of self-modifying code, code that mutates during runtime, is a complicated but impressive undertaking that creates an incredibly robust obfuscating system. While there is a great amount of research that is still ongoing, the preliminary results of this subject suggest that the application of self-modifying code to obfuscation may yield self-maintaining software capable of healing itself following an attack.

Keywords: self-modifying code, autonomous software, obfuscation

Achieving Obfuscation Through Self-Modifying Code: A Theoretical Model

Executive Summary

Computing systems and modern networks have completely revolutionized the world in a startlingly short amount of time. Information is available at the touch of a button, and cellular devices and Internet of Things (IoT) devices have brought connectivity into many facets of modern life. In usual software development practices, programming is relatively straightforward and follows general standards. One of the methods of programming that does not follow these standards, and is widely discouraged and very rarely used, is *self-modifying code*. Often villainized for its incredibly complex and difficult nature, self-modifying code is exactly what it sounds like – code that can modify itself during execution. This practice is fraught with difficulties and dangerous side effects, so much so that high-level languages do not allow for its use (it can only be accomplished using assembly code). However, despite the difficulties, self-modifying code has many extraordinary uses, with research being done into its application in self-healing networks and autonomous software. Most commonly, however, it is used for obfuscation.

Protecting information stored online is one of the most critical jobs in existence. There are many ways that security administrators work to protect information, with two popular and notably similar methods being *encryption* and *obfuscation*. Encryption is a method of transforming data into an unreadable code using keys, while obfuscation attempts to render the data unreadable without using a key or a code. The primary difference is that, in encryption, an attacker cannot decrypt the data without the key, even if he knows the encryption algorithm being used, while in obfuscation, the data can be understood if the attacker knows the algorithm used to obfuscate it. Essentially, obfuscation is a faster but less secure method of hiding data. There are many methods of obfuscating data; one of the most popular methods involves passing the binary digits of the data

through an exclusive OR (XOR) operation along with a generated key string to flip some of the bits and render the data into an unrecognizable state.

A large amount of research is being done into creating self-obfuscating software applications using self-modifying code. This thesis proposes a new theoretical method, which uses an XOR obfuscation algorithm that reapplies the obfuscation every few minutes using a different XOR input string with a key generated from a random byte of information gained from one of the memory registers. This way, the obfuscated instructions are constantly changing, making it extremely difficult – nearly impossible – for an attacker to decipher the algorithm being used. Although this algorithm is purely theoretical and requires a large amount of development and testing, it has the potential to offer a robust and easy-to-maintain method of protecting data.

Introduction to Self-Modifying Code

In the realm of computer science and software development, certain subjects are generally avoided because of their inherent difficulty. One such area is the design and implementation of self-modifying software, which is code that can dynamically modify itself at runtime (Cai, Shao, & Vaynberg, 2007). Its unpopularity is mostly due to its extremely complicated nature in all areas, including design, implementation, and debugging. However, when properly implemented, self-mutating code can provide startling improvements in code optimization and security practices and may provide the foundation for a software system that can independently identify a security breach, fix the problem, and modify its own code for future security.

Self-modifying code has multiple definitions, and clarifying the meaning behind the term is an important step in order to fully grasp the concept. One definition is something that is seen frequently throughout computer programming, where the code being executed depends on variables that are entered at runtime. For example, consider the code segment below:

```
1   #include <iostream>
2
3   using namespace std;
4   int main()
5   {
6       int counter;
7
8       cout << "Please enter a number: ";
9       cin >> counter;
10
11      for (int i = 0; i < counter; i++)
12      {
13          cout << "Iteration " << i + 1 << endl;
14      }
15
16      system("pause");
17      return 0;
18  }
```

Figure 1. Basic `for` loop code.

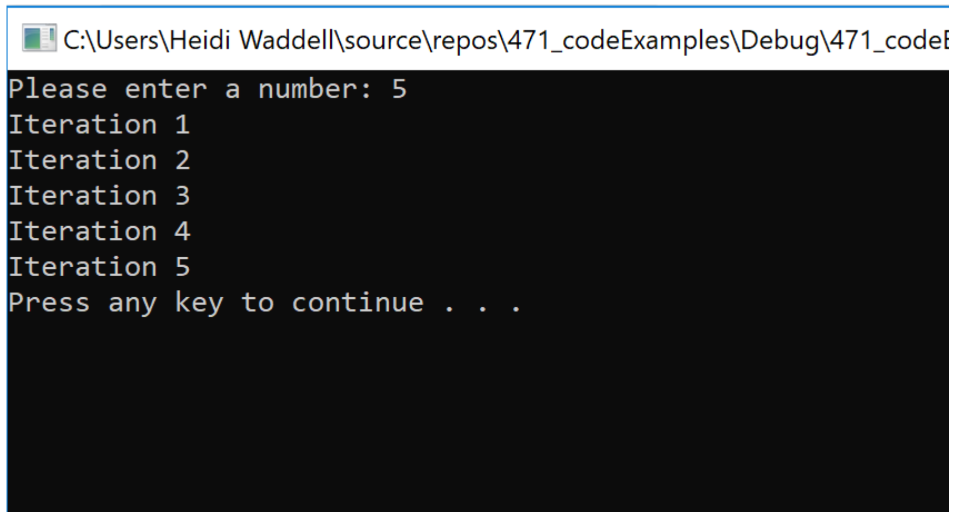

Figure 2. `For` loop code results.

In this example of a basic `for` loop, the number of times that the loop executes is an unspecified variable that is entered at run-time by the user of the system. Since this causes the code to be modified during execution, it can be considered an example of self-modifying code. However, for the purposes of this project, this is not the kind of code being investigated. Rather, when self-modifying code is referenced, it is referring to the modification of specific instructions by causing the compiler to store instructions as data:

> Writes and reads of the data memory both occur in the memory stage [of the stages taken by the processor to execute instructions]. By the time an instruction reading memory reaches this stage, any preceding instructions writing memory will have already done so. On the other hand, there can be interference between instructions writing data in the memory stage and the reading of instructions in the fetch stage, since the instruction and data memories reference a single address space. This can only happen with programs containing *self-modifying code*, where instructions write to a portion of memory from which instructions are later fetched. Some systems have complex mechanisms to detect

> and avoid such hazards, while others simply mandate that programs should not use self-modifying code. (Bryant & O'Hallaron, 2010, p. 471)

This description presupposes the use of pipelining, a commonly used practice in modern computing where multiple instructions are executed at the same time.

Source Code Obfuscation

Data protection and information security has rapidly become one of the most in-demand fields in existence. With the vast amount of data stored online via networks and source code, ensuring the confidentiality, integrity, and availability of the information is a crucial task to maintain the safety of internet data. Ideally, source code and data would be completely unreachable by an unauthorized user; however, there is no such thing as safe computing, and it is inevitable that breaches to a system will occur. Therefore, defense schemes aim to create *defense-in-depth* – multiple layers of security on each aspect of computation that aim to achieve security (preventing a breach from occurring) and resiliency (minimizing the damage after a breach). One of the most popular methods of achieving resiliency is to mask its true meaning or value, rendering it useless to the attacker.

The most popular means of hiding data or source are through *encryption* and *obfuscation*. While these topics may seem like the same concept, they are subtly different in method and overall goal. Encryption is defined as transforming the data into a code using a key, which then also requires a key to translate the data back into its original form. Obfuscation, however, does not require a key to translate the data into a predictable code; instead, obfuscation renders the data into an unreadable form that still accomplishes the same task:

> The code obfuscation is a mechanism for hiding the original algorithm, data structures or the logic of the code, or to harden or protect the code (which is considered as intellectual property of the software writer) from the unauthorized reverse engineering process. In general, code obfuscation involves hiding a program's implementation details from an adversary, i.e. transforming the program into a semantically equivalent (same computational effect program, which is much harder to understand for an attacker). (Behera & Bhaskari, 2015, p. 757)

Obfuscation is especially useful to guarding against source code tampering, malicious reverse engineering, and the theft of intellectual property.

Typically, encryption is performed on data, and obfuscation is performed on source code. In this application, the goal of obfuscating source code is to hide what the code accomplishes. This way, if an attacker gains access to the functions, it will be unclear what the purpose or variables of the code is. For example, consider the code below:

```
/*
  LEAST LIKELY TO COMPILE SUCCESSFULLY:
  Ian Phillipps, Cambridge Consultants Ltd., Cambridge, England
*/

#include <stdio.h>
main(t,_,a)
char
*
a;
{
    return!

0<t?
t<3?

main(-79,-13,a+
main(-87,1-_,
main(-86, 0, a+1 )

+a)):

1,
t<_?
main(t+1, _, a )
:3,

main ( -94, -27+t, a )
&&t == 2 ?_
<13 ?

main ( 2, _+1, "%s %d %d\n" )

:9:16:
t<0?
t<-72?
main( _, t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#;\
#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/+k#;\
q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){nl]!/n{n#'; \
r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;\
{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:
t<-50?
_==*a ?
putchar(31[a]):

main(-65,_,a+1)
:
main((*a == '/') + t, _, a + 1 )
:

0<t?

main ( 2, 2 , "%s")
:*a=='/'||

main(0,

main(-61,*a, "!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry")

,a+1);}
```

Figure 3. Twelve Days of Christmas obfuscated code.

While totally unreadable, this code outputs all 12 verses of the Christmas song *The Twelve Days of Christmas* (this particular code won the International Obfuscated C Code Competition in 1998) (International Obfuscated C Code Winners 1988 - Least likely to compile successfully, n.d.).

Computer Architecture

One of the chief reasons for the unpopularity of self-modifying code is the extreme level of difficulty in successfully implementing it. To fully understand why this is the case, as well as how to properly implement self-mutating code, an understanding of computer hardware and modern processes must be established. Modern computer architecture is based on the von Neumann architecture, which contains multiple hardware units that work together to execute software:

> The von Neumann computation model is the most common and commercially successful model to date. The main characteristic of this model is a single separate storage structure (the memory) that holds both program and data. Another important characteristic is the transfer of control between addressable instructions, using a program counter (PC). The transfer is either implicit (auto-increment of PC) or through explicit control instructions (jumps and branches, assignment to PC). It is for this reason that the von Neumann model is commonly referred to as a control flow model. (Yazdanpanah, Alvarez-Martinez, & Jimenez-Gonzalez, 2014, p. 1490)

A von Neumann system is made up of many components. The *central processing unit* (CPU) is the cornerstone of the hardware system, and it is the unit that interprets instructions written by the software to manipulate and compute data. For this project, the key components to understand are *registers*, set locations inside of the CPU that contain small amounts of memory. Data can be moved from memory into one of these registers, where manipulations are made to the value, and then the new value is transferred back into memory. *Memory* holds a program being executed, and *input/output* (I/O) devices can read in information from the user and write it out to either a disk or a screen. Electrical conduits called *buses* ferry bytes between all the

different components, and the number of bits that the buses can carry determines the word size of the machine. Modern day machines typically implement either 4-byte (32-bit) or 8-byte (64-bit) word sizes.

Two crucial sections of memory to understand are the heap and the stack, two of the most fundamental data structures that computers implement. The heap is the memory section that provides additional memory for an application when requested, and the stack contains the instruction sequence, arguments, and variables that run during execution. This can be seen in Figure 4 below:

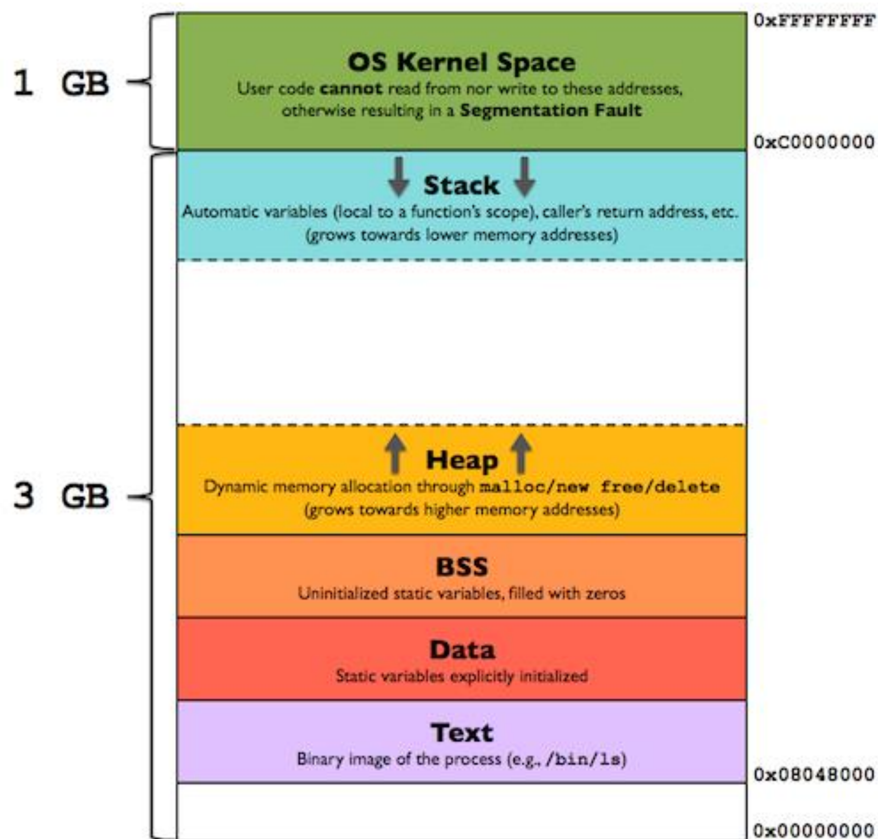

Figure 4. How a program appears in main memory.

Although depictions of computer architecture divide main memory and control structures into sections and divisions, this organization is purely logical and not physical (Mavrogiannopoulos,

Kisserli, & Preneel, 2011). This flat layout causes a lack of distinction between code and data, and this allows applications to store data in memory that will later be interpreted as instructions.

Ultimately, all information processed by computer systems are binary digits, and can be either a 0 or a 1. These individual 0's and 1's, known as *bits* (short for 'binary digits'), represent all the information on a system. This includes programs stored and running in memory, user data, network transmissions and information, and every other piece of information stored in internal or external memory. The binary nature of bits is because they represent electrical voltage, with 0's being a low voltage and 1's being a high voltage. While specific voltage capacity meters vary, values of 0 – 0.3 MHz are interpreted as a 0, and values of 0.7 – 1 MHz are interpreted as a 1. Since all information is represented as bits, the only differentiation between the different values is the context in which they are viewed: "The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instructions" (Bryant & O'Hallaron, 2010, p. 39). How are these bits turned into useful information?

High-level programming languages, such as C++ and Java, are modified several times by the computer system. First, the compiler translates the program (to which the necessary headers have been added by the preprocessor) into assembly language, a low-level language that represents the commands using English characters and semi-readable mnemonics. From the assembly language, the assembler generates object code, represented by hexadecimal values, and finally, the object code is turned into machine code, long strings of binary digits that the computer hardware interprets as electrical signals. The vast majority of modern programming is accomplished by high-level programming languages, which are readable by humans and can

provide safeguards and debugging for code developers. However, most of these languages use said safeguards to prevent self-modifying code; therefore, writing a self-modifying program is best developed using assembly language, which does not contain code guidelines and communicates directly with the physical hardware of the system. This is part of what makes self-modifying code so difficult; although assembly language uses English characters arranged into somewhat comprehensible words, it is highly tedious to develop in as it requires developers to move values individually to and from each register, specify specific memory locations, and to interact directly with the stack.

Assembly Language

The first computers, developed in the 1950s, were vast, complicated machines that could carry out only one computation at a time. Initially, they were powered by thousands of electrical devices called vacuum tubes that controlled electric current flow, but these were soon replaced by transistors, semiconductor devices which generated less heat and led to faster computing. Improving software performance was a slower process than improving the hardware, however; but it was John von Neumann's development of binary instructions that laid the groundwork for modern-day software. When von Neumann formulated the architecture of modern processors, he organized it so that a string of bits would be used to encode both the instructions and the data of the program, leading to the development of modern assembly language. Paul Dunne (n.d.). explains:

> Following this approach a machine that operated on, say 16 bit *words*, the memory locations that held the program would have the instructions interpreted as follows: the first few bits (4 for example) would indicate a particular operation (ADD, STORE, LOAD etc) and the remaining bits (12 in this case) would indicate where the data for the

> operation was stored in memory. For a such a program to be executable by the computer, however, the binary pattern corresponding to each individual instruction would have to be entered into the memory. A typical application program for a complex scientific calculation might break down into 200 or more such instructions and so to carry out the calculation 3200 0s and 1s would have to be produced and loaded into memory. (para. 2)

Initially, commands were issued using punched cards representing individual bits, but this was soon replaced by assembly language, a low-level language slightly higher than the machine code. This was then built upon with the advent of high-level languages, beginning with FORTRAN (FORmula TRANslation) and then COBOL (Common Or Business Oriented Language), which enabled programmers to create more efficient code with less errors. Assembly language still remains in use, as modern programming languages are disassembled into assembly code before object and machine code; however, it is very rare to develop software directly in assembly language anymore. This process of translation between languages is diagrammed in Figure 5:

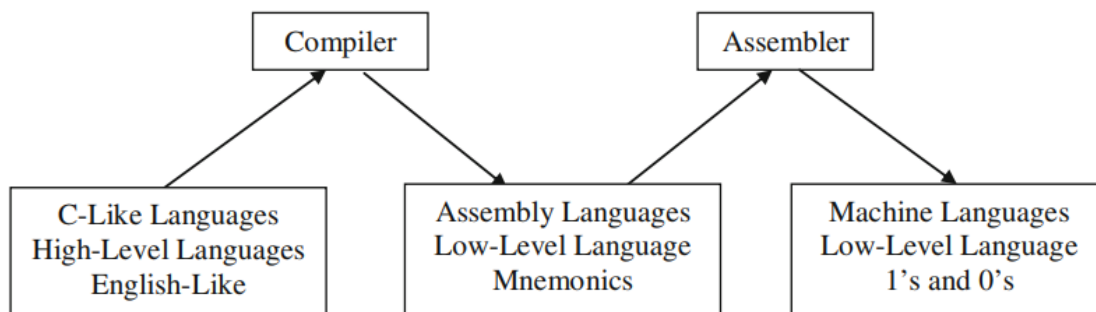

Figure 5. High-level language and assembly language translation to machine language.

Despite the difficulties involved in developing assembly language, there are several advantages that make it worth the hassle in certain situations: "The advantage of programming in

assembly language over a high-level language is that one can gain a very detailed look at the architecture of a computer system and write very efficient programs, in terms of both increasing speed and saving memory" (Streib, 2011, p. 1). Because assembly programs deal directly with the computer hardware, registers and memory locations can be directly accessed, thus making its development much more dangerous but also much more powerful.

To demonstrate the syntax and setup of assembly language, a comparison between a program written in C++ and the same program written in assembly code is shown below:

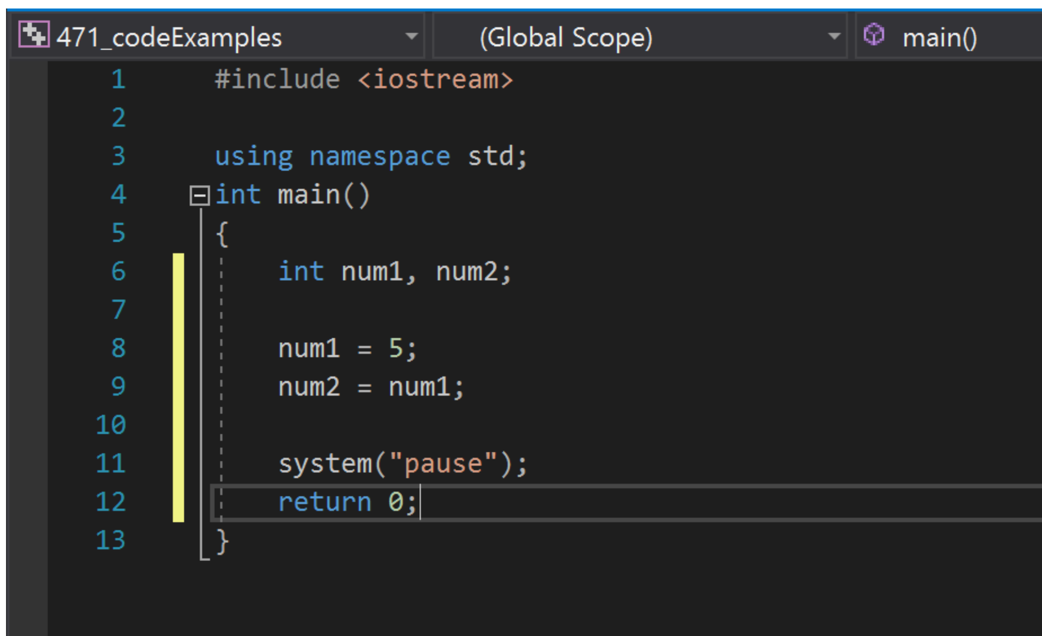

Figure 6. A basic C++ program.

```
main.asm*
1   .386
2   .model flat, stdcall
3   .stack 100 h
4   ExitProcess PROTO, dwExitCode: DWORD
5
6   .data
7   num1    sdword ?    ; first number
8   num2    sdword ?    ; second number
9
10  .code
11  main PROC
12      mov num1, 5     ; initialize num1 with 5
13      mov %eax, num1  ; load eax with contents of num1
14      mov num2, %eax  ; store eax in num2
15      ret
16
17  main ENDP
18  END main
19
```

Figure 7. Assembly language program of Figure 4[1].

Upon first inspection, the assembly code is very difficult to understand, as its syntax is mnemonic in nature and does not use full English words the way high-level languages do. The first thing to understand is the difference between directives and instructions. Instructions are implemented by the CPU, and can be seen in the commands on lines 7-8 and 12-15. Directives, however, tell the assembler what to do, and can be seen in lines 1-4, 6, 10, and 17-18. For example, the `.386` directive at the beginning of the program instructs the assembler that the program will be run on an Intel 386 or newer processor (used on 64-bit systems), and the `.stack 4096` directive tells the assembler how large the stack will be (in this case, 100 hexadecimal bytes).

[1] This program is written in the Microsoft Macro Assembler (MASM), an x86 assembler implemented on Intel processors.

In the x86 architecture, there are a total of 16 registers that the system uses to manipulate data (although 64-bit architectures contain additional, rarely used registers), and are denoted in the Microsoft Macro Assembler (MASM) with a percent sign (%). Eight of these sixteen are designated as general-purpose registers, and are used to temporarily save data inside the processor:

1. Accumulator register (AX)
2. Counter register (CX)
3. Data register (DX)
4. Base register (BX)
5. Stack Pointer register (SP)
6. Stack Base Pointer register (BP)
7. Source Index register (SI)
8. Destination Index register (DI)

Six of the remaining registers are called segment registers, and typically do not change value during the execution of a program:

1. Stack Segment register (SS)
2. Code Segment register (CS)
3. Data Segment register (DS)
4. Extra Segment register (ES)
5. F Segment register (FS)
6. G Segment register (GS)

The two remaining registers are the Instruction Pointer register (IP), which contains the address of the next instruction that the CPU will execute, and the Flags register, which is set to a specific value in certain cases (for example, if the result of an operation results in a value too large for the register to represent, the 'Overflow' flag is set). In 64-bit registers, the register acronyms are prefixed by an 'R', and in 32-bit registers, they are prefixed by an 'E' (in other words, the ax register is referred to as `%rax` or `%eax`).

With this basic understanding of assembly language now established, the instructions in the MASM program above can be examined. The `num1 sdword ?` and `num2 sdword ?` instructions create two variables, named `num1` and `num2`, that are 32-bits in length (the size of an sdword). In the main program, denoted by the `main proc` (which stands for 'main procedure'), the value of 5 is moved into the `num1` variable by the command `mov num1, 5`. The `mov %eax, num1` command then loads this variable into the `%eax` register, and then stores this register into `num2` with the command `mov num2, %eax`. The `ret` command returns the value of zero, signaling the end of the program.

Application

With a firm understanding of computer architecture and assembly language, self-modifying code can now be understood and implemented. The applications of dynamically mutating code range from software optimization and independent code adaptation to security and obfuscation. It has been fully established that there is no such thing as completely secure software; no matter how many security devices are put in place, new advances in malware will always pose new threats and exploit unforeseen vulnerabilities in the software. Security, therefore, is a living process that must be constantly monitored and maintained. Current security

processes are done largely by humans or with the use of a semi-automated process. The application of self-modifying code, however, could play a key role in the creation of a full-automated, independent software system capable of adapting to new security situations and healing itself from breaches and hardware failures (Rschudin & Yamamoto, 2006).

While there are many applications in a variety of programs, most common of which are viruses and malware that can replicate themselves over and over again, the application that will be discussed in this project will be overwriting data and source code for obfuscation and security. Masking the machine code, and thus making software unintelligible to the attacker, provides a layer of security that will be crucial for the probable (possibly inevitable) event of a security breach, allowing the system time to find and correct the vulnerabilities without losing all data to the attacker (Mavrogiannopoulos, Kisserli, & Preneel, 2011). When used in this way, if the program being executed senses a network breach, it can execute a loop that will overwrite the data section of the program with bogus values. This renders the data unusable to the intruder, although it may also destroy the data permanently. Overwriting the data using a specific encryption technique achieves obfuscation, which renders the variables unusable temporarily but allows for them to be decrypted back into their original values. The other popular method of obfuscation, which will be explored later, is overwriting the instructions themselves with a different command. As this obfuscation does not change the output of the application, it can be carried out at certain intervals over and over again as the program runs, thus causing the assembly commands to be constantly re-obfuscated. This adds a great level of complexity to the algorithm, making it drastically harder to an intruder to decipher.

One way to achieve data obfuscation, purposefully overwriting data values, is by causing bogus values to overwrite the values in the stack. As was discussed previously, computer

memory is shown and explained with different sections for data and program executions, but in fact this distinction is purely logical and is not actually achieved in the underlying architecture. Thus, it is possible to intentionally overwrite values in the data section directly using instructions in the code section. As an example, consider an assembly language program with stack that begins at `.pos 0x100`. As values are pushed, the stack grows 'upward' towards the smaller memory addresses. This is illustrated below:

Figure 8. Basic assembly program.[2]

[2] This program is written in a Y86 Simulator. Y86 is similar to the x86, but is used as an introduction to assembly language development as it contains fewer instructions and simpler syntax. It is used purely for educational purposes.

In this example, the assembly code (shown in the far-left column) sets up the base pointer register `%ebp` (which contains the address of the base of the stack) and the stack pointer register `%esp` (which contains the address of the top of the stack), moves the value of 5 into the `%eax` register, and then pushes this value onto the stack. At the bottom, the `.pos 0x100` command is an assembler directive that causes the `Stack` variable to be created at that position in memory. In the far-right column, the values of each position in memory are shown, along with the position of the base and top of the stack[3]. As this column illustrates, the memory locations grow downwards, but the stack values grow upwards (towards the smaller memory addresses). In order to overwrite the values on the stack, an assembly command would cause the value in a register to be written to the position in memory where the stack resides. High-level programming languages do not allow for this capability, as it is incredibly dangerous and can have disastrous consequences when it occurs unintentionally. When used intentionally, however, obfuscation and data protection can be achieved directly and efficiently.

While masking the values and variables of an application is a popular application of obfuscation, the primary method of obfuscation involves masking the source code itself. Thus, instead of overwriting the values in the stack, the commands themselves are hidden and obscured. This method of obfuscation is evaluated closely in the next section, and a theoretical algorithm for achieving source code obfuscation with self-modifying code is proposed and outlined.

[3] Please note, this free version of the Y86 simulator contains an error that causes the `%ebp` and `%esp` registers to point to the wrong position in memory. The correct position is 3 bytes above.

Methods of Source Code Obfuscation

XOR Obfuscation

One of the most popular methods of obfuscation in use is known as *XOR obfuscation*, which takes its name from the Boolean algebra truth table known as *exclusive OR* (XOR) (Kissel, 2005). Boolean truth tables take inputs of binary digits (i.e., either 0 or 1), and output a binary digit based on the combination of the input. An *OR* table outputs a 1 if either one of the two bits input is a 1; it's derivate XOR outputs a one if either of the two bits is a 1, but not both. This is displayed in the following tables:

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

Figure 9. OR Table.

Input		Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 10. XOR Table.

In computer science, the XOR operator is a bitwise operator that takes two binary strings as inputs and creates a resulting binary string by performing the XOR operation on all the bits. For example, the XOR results of the string *01011101* and the string *1110001* is *10111100*. When applied to obfuscation, a key string is generated that is then passed to an XOR function along with the code, and the original binary digits are then overwritten with the results of this operation. This method of obfuscation is extremely similar to encryption, which always involves

the use of keys, but is still classified as obfuscation because of the method and goal of the operation: “XOR(255) has the advantage of being fast (it typically executes in less than 1 clock cycle on modern architectures), reversible, and can be performed in-place. XOR(255) has the additional property of leaving a file’s entropy unchanged, allowing processed data to remain invisible to tools that search for encrypted data using entropy techniques” (Zarate, Garfinkel, Herrernan, Gorak, & Horas, 2014, p. 1).

Achieving XOR obfuscation is accomplished in a variety of ways. The most basic method is performing an XOR operation on two bytes, and keeping the result. For example, consider obfuscating the letter ‘J’ by XORing it with the letter ‘v’. The first step is to find the American Standard Code for Information Interchange) ASCII values that correspond to these letters by consulting a table such as the one below. ASCII values are universal codes, displayed in the hexadecimal number system, that the computer translates into numbers and symbols:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Figure 11. ASCII Table.

By this table, ‘J’ is equal to the hexadecimal value of *0x4A* and ‘v’ is the hexadecimal value of *0x76*. Converting these values into binary values results in *01001010* and *01110110*. Performing an XOR operation on these bytes results in the value of *00111100*. Turning this byte back into a hexadecimal value results in *0x3C*, which by referencing the table results in the symbol ‘<’.

Another method of obfuscation, which is more advanced and complicated, combines machine language with assembly language. This is possible due to the extremely basic nature of assembly commands. This method is demonstrated below:

Program-5 (Original Code)	Program-5 (Obfuscated Code)

.model small
.code
Mov DH, 85
Mov AH, 2
Mov CL, 80
Mov DX, 35537
Int 21h
Mov DL, DH
Int 21h
Mov AH, 76
Int 21h
end
	.model small
	.code
	Mov DH, 85
	Mov AH, 2
	Mov CL, 80
	db 10111010b
	Mov DL, CL
	Int 21h
	Mov DL, DH
	Int 21h
	Mov AH, 76
	Int 21h
	end

Figure 12. Combining machine and assembly languages.

Since assembly language is one step above machine language, the machine code ‘10111010’ is the machine translation of ‘Mov DX’. Converting 35537 into binary results in ‘1000101011010001’, in the format of ‘Mov-dw-11-DL-CL’ (which results in Mov Dl, CL) (Behera & Bhaskari, 2015).

Other Methods

Although XOR obfuscation is arguably the most popular obfuscation method in use today, several other methods exist. *Dead-Code Insertion* is a method that simply adds commands that do not accomplish anything, making the malicious actor think that the code is

accomplishing a different task. *Instruction Subroutines* replace original commands with other commands that accomplish the same task (for example, *xor* can be replaced by *sub*, and *mov* is equivalent to *push*). *Code Transportation*, possibly the most difficult method of obfuscation, rearranges the commands of an applications in a way that does not affect the output (Iliev, 2017).

Achieving Obfuscation through Self-Modifying Code: A Theoretical Algorithm

One of the most popular applications of self-modifying code is in the realm of obfuscation. However, the extremely difficult nature of dynamically modifying code has resulted in a lack of research and development in the subject. Therefore, the goal of this senior thesis is to propose an algorithm, written for assembly language, that will present a means of modifying commands during execution to obfuscate the commands. The general flow of the algorithm is as follows: first, variables *key* and *counter* are declared, and counter is set to 0. *key* is an 8-bit (1-byte) value that contains a binary string gained from the stack pointer (SP) register. As discussed previously, the SP register is one of the eight general-purpose registers used to temporarily save data inside of the processor, and its purpose is to hold the address of the most recent value on the stack. Therefore, the value contained in it is constantly changing as various programs are executed, making the resulting value of k*ey* nearly impossible to discern by an attacker. Using this value may cause security concerns; however, since this value is only being used as the key by which to obfuscate the desired information, these concerns are minimal. After these values are set up, the necessary data that the code is manipulating is loaded into the stack. In a real-life application, these values could represent usernames, passwords, email addresses, or other necessary data. These values are then manipulated according to the needs of the program, but after every command is executed, the *counter* variable is incremented. Once *counter* is equal to *10*, *key* will be regenerated according to the current value in the SP register, and the command to be obfuscated (which for the sake of this program will be a *mov*

command) is passed through an *XOR* operation with *key*. The original command will then be replaced with this new value. This algorithm is explained in pseudocode in the following figure:

```
Variables
    Key
    Counter

Pseudocode
    counter = 0
    key = toString(getTime())

    Move data into registers
    Move data into stack

    Main:
        For (counter; counter<5; counter++)
            Manipulate data
            Increment counter
            If counter == 5
                Invoke newKey function

    newKey:
        key = toString(getValueInSP())
        xor 'move data into stack' command with key
        Set 'move data into stack' command as xor result
        return
```

Figure 13. Proposed algorithm.

To properly understand how this algorithm works, consider the result of obfuscating a *mov* command. In assembly language, *mov* is used to move data values between registers and memory. "This instruction has two operands: the first is the destination and the second specifies the source. Some examples of mov instructions using address computations are:

mov eax, [ebx] ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx ; Move the contents of EBX into the 4 bytes at memory address var. (Note, var is a 32-bit constant).
mov eax, [esi-4] ; Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl ; Move the contents of CL into the byte at address ESI+EAX

mov edx, [esi+4*ebx] ; Move the 4 bytes of data at address ESI+4*EBX into EDX

(x86 Assembly Guide, 2018, para. 11). In the assembly command *mov al, 61h*, the value of 61 in hexadecimal digits (97 in the decimal system) is loaded into the AL register, and this can be broken down into four specific parts of binary digits. The opcode of the *mov* command is *1011*; the single bit *0* is used to specify if the data is a byte or a full-size of 16/32 bits; the binary identifier for *AL* is *000;* and the binary representation is *0x61* is *01100001*. Adding these values together results in the complete binary opcode of *10110000 01100001* (Computer architecture and assembly language, n.d.).

To obfuscate this command, the algorithm must first generate a key based on the location of the current instruction on the stack, contained in the SP register. For this example, the data inside of SP is *10010001*, and therefore this value is copied into *key* giving it a value of *10010001*. This key will first be passed through an XOR operation with the first byte of the opcode, resulting in the value of *00100001*. The same operation will then be applied to the second byte of the opcode, giving a value of *11110000*. With the new obfuscated version of the opcode now generated, the program must now overwrite the original command with the new opcode. This can be accomplished in several ways; the command itself can be overwritten directly, or new commands can be generated that skip the original command and instead execute from the newly generated one. For example, consider this self-modifying code example:

```
// Define the size of the self-modifying function.
#define SELF_SIZE
((int) x_self_mod_end - (int) x_self_mod_code)

// Start of the self-modifying function. The naked
// qualifier supported by the Microsoft Visual C compiler
// instructs the compiler to create a naked
// Assembly function, into which the compiler
// must not include any unrelated garbage.
__declspec(naked)
int x_self_mod_code(int a, int b) {
    __asm{
        begin_sm:                   ; Start of the self-modifying code.
        MOV EAX, [ESP + 4]          ; Get the first argument.
        CALL get_eip                ; Define the current position in memory.
        get_eip:
           ADD EAX, [ESP + 8 + 4]   ; Add or subtract the second argument
                                    ; from the first one.
           POP EDX                  ; EDX contains the starting address of
                                    ; the ADD EAX, ... instruction.
           XOR byte ptr [edx], 28h  ; Change ADD to SUB and vice versa.
           RET                      ; Return to the parent function.
    }
}

x_self_mod_end()

{/* End of the self-modifying function */ }

main() {
    int a;
    int (__cdecl *self_mod_code)(int a, int b);
    // Uncomment the next string to make sure
    // that self-modification under Windows is
    // impossible (the system will throw an exception).
    // self_mod_code(4, 2);

    // Allocate memory from the heap (where
    // self-modification is allowed). With the same success,
    // it is possible to allocate memory in the stack:
    // self_mod_code[SELF_SIZE];
    self_mod_code = (int (__cdecl*)(int, int)) malloc(SELF_SIZE);

    // Copy the self-modifying code into the stack or heap.
    memcpy(self_mod_code, x_self_mod_code, SELF_SIZE);

    // Call the self-modifying procedure ten times.
    for (a=1; a<10; a++) printf("%02X",self_mod_code(4,2)); printf("\n");
}
```

Figure 14. Self-modifying code example.

When the *get_eip* function is called in the first function, the address of the *ADD* instruction is popped from the stack and replaced by the new one generated by the command *XOR byte ptr [edx], 28h*: "Self-modifying code replaces the ADD machine code with SUB and

sub with ADD ; therefore, calling self_mod_code in a loop returns the following sequence of numbers : 06 02 06 02... , thus confirming successful completion of self-modification" (Principles of building self-modifying code, n.d., para. 7).

Conclusion

High-level programming languages are created for a number of reasons. In early years, computer programming was done completely with assembly language, but the tedious nature of the logic and syntax, coupled with the lack of debuggers or data protection, caused many issues for developers. Thus, creating languages like Java, Python, and the C variations led to more efficient programming that could create software much easier and with far fewer errors. Now, assembly language is rarely used, and is usually only taught to rising developers as a means of understanding computer architecture. However, harnessing the power of assembly language can lead to dynamic programs with a variety of applications, with self-mutating code being one of the most common applications. Autonomous software may not be practical for every application, but in applications like embedded networks or satellite technology that cannot be accessed in real-time, it may provide answers to the problem of security (Rschudin & Yamamoto, 2006).

Dynamic and self-modifying code may be difficult and time-consuming, but applying it to the topic of security may lead to exciting developments and programs that can automatically protect sensitive data from intruders. In addition to obfuscation, self-mutating code can also be used to directly create new instructions, modify source code, and even create and execute a dynamic program that can write itself without human interaction (Becker, 2013). Other possible applications of self-modifying code include software that can automatically optimize itself for maximum efficiency and self-healing software that can develop its own updates and source code

modification. When coupled with the emerging capabilities of artificial intelligence (AI), the future of self-modifying code is limitless.

References

Ansel, J., Marchenko, P., Erlingsson, U., & Taylor, E. (2011). Language-independent sandboxing of just-in-time compilation and self-modifying code. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, *11*, 355-366.

Balachandran, V., & Emmanuel, S. (2013). Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEE Transactions on Information Forensics and Security*, *8*(4), 669-681.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001). On the (im)possibility of obfuscating programs. *Annual International Cryptology Conference*, *2139*, 1-18.

Becker, K. (2013, January 27). Using artificial intelligence to write self-modifying/improving programs. Retrieved from http://www.primaryobjects.com/2013/01/27/using-artificial-intelligence-to-write-self-modifying-improving-programs/

Behera, C. K., & Bhaskari, D. L. (2015). Different obfuscation techniques for code protection. *Procedia Computer Science*, *70*, 757-763.

Bitansky, N., & Vaikuntanathan, V. (2015). Indistinguishability obfuscation from functional encryption. *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, 171-190.

Blazy, S., Laporte, V., & Pichardie, D. (2016). Verified abstract interpretation techniques for disassembling low-level self-modifying code. *Journal of Automated Reasoning*, *56*, 283-308.

Brecak, J. (2004, November 23). Self modifying C code. Retrieved from https://web.archive.org/web/20100717072236/http:/public.carnet.hr/~jbrecak/sm.html

Brunton, F. (2015). *Obfuscation: A user's guide for privacy and protest*. Cambridge, MA: The MIT Press.

Bruschi, D., Martignoni, L., & Monga, M. (2007). Code normalization for self-mutating malware. *IEE Security & Privacy*, *5*(2), 46-54.

Bryant, R., & O'Hallaron, D. (2010). *Computer systems: A programmer's perspective*. Harlow, United Kingdom: Pearson.

Cai, H., Shao, Z., & Vaynberg, A. (2007). Certified self-modifying code. *ACM SIGPLAN Notices - Proceedings of the 2007 PLDI Conference*, *42*(6), 66-77.

Figure 11. ASCII Table. Reprinted from *Nowhere to hide: three methods of XOR obfuscation*, by Cannel, J. (2016). Retrieved from https://blog.malwarebytes.com/threat-analysis/2013/05/nowhere-to-hide-three-methods-of-xor-obfuscation/

Figure 12. Combining machine and assembly languages. Reprinted from *Nowhere to hide: three methods of XOR obfuscation*, by Cannel, J. (2016). Retrieved from https://blog.malwarebytes.com/threat-analysis/2013/05/nowhere-to-hide-three-methods-of-xor-obfuscation/

Collberg, C., & Nagra, J. (2010). *Surreptitious software obfuscation, watermarking, and tamperproofing for software protection*. Upper Saddle River, NJ: Addison-Wesley.

Computer architecture and assembly language [PowerPoint slides]. (n.d.). Retrieved from https://www.cs.bgu.ac.il/~caspl162/wiki.files/PS01_162[2].pdf

Dang, B., Gazet, A., Bachaalany, E., & Josse, S. (2014). *Practical reverse engineering: x86, x64, arm, windows kernel, reversing tools, and obfuscation*. Indianapolis, IN: John Wiley and Sons.

Deitel, P., & Deitel, H. (2014). *C++ How to Program*. Edinburgh, Scotland: Pearson.

Dovland, J., Johnsen, E. B., Owe, O., & Yu, I. C. (2015). A proof system for adaptable class hierarchies. *Journal of Logical and Algebraic Methods in Programming*, *84*(1), 37-53.

Dunne, P. (n.d.). Making computing easier: programming languages. Retrieved from http://cgi.csc.liv.ac.uk/~ped/teachadmin/histsci/htmlform/lect6.html

Goldwasser, S., & Rothblum, G. (2014). On best-possible obfuscation. *Journal of Cryptology*, *4392*, 480-505.

Iliev, K. (2017, August 31). Top 6 advanced obfuscation techniques hiding malware on your device. Retrieved from https://sensorstechforum.com/advanced-obfuscation-techniques-malware/

Figure 3. International Obfuscated C Code Winners 1988 - Least likely to compile successfully. (n.d.). Reprinted from *5th International Obfuscated C Code Contest* (n.d.). Retrieved from http://www.ioccc.org/years.html#1988_phillipps

Kissel, Z. (2005). Obfuscation of the standard XOR encryption algorithm. *Crossroads, 11*(3), 6-6.

Ligh, M. H. (2011). *Malware analyst's cookbook and dvd: Tools and techniques for fighting malicious code*. Indianapolis, IN: Wiley.

Mavrogiannopoulos, N., Kisserli, N., & Preneel, B. (2011). A taxonomy of self-modifying code for obfuscation. *Computers & Security*, *30*(8), 679-691.

Morse, G. (2018). Pure infinitely self-modifying code is realizable and Turing-complete. *International Journal of Electronics and Telecommunications*, *64*(2), 123-129.

Figure 14. Principles of building self-modifying code. (n.d.). Reprinted from *Flylib*, (n.d.). Retrieved from https://flylib.com/books/en/1.444.1.62/1/

Rschudin, C., & Yamamoto, L. (2006). Harnessing self-modifying code for resilient software. *Lecture Notes in Computer Science*, *3825*, 197-204.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting software through obfuscation: can it keep pace with progress in code analysis? *ACM Computing Surveys*, *49*(1), 1-37.

Shan, L., & Emmanuel, S. (2011). Mobile agent protection with self-modifying code. *Journal of Signal Processing Systems*, *65*, 105-116.

Streib, J. (2011). *Guide to assembly language: A concise introduction*. London, England: Springer-Verlag.

Figure 5. High-level language and assembly language translation to machine language. Reprinted from *Guide to assembly language: A concise introduction*, by Streib, J. (2011).

Figure 4. In-memory layout of a program (process). Reprinted from *Computer Science, Research, Data, and Code*, by Tolomei, G. (n.d.). Retrieved from https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Tully, S. (2013, December 29). Writing a self-mutating x86_64 c program. Retrieved from https://shanetully.com/2013/12/writing-a-self-mutating-x86_64-c-program/

Viega, J. (2003). *Secure programming cookbook for C and C++*. Sebastopol, CA: O'Reilly.

x86 Assembly Guide. (2018). Retrieved from https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Xu, H. (2016). Assessing the security properties of software obfuscation. *IEEE Security and Privacy Magazine*, *14*(5), 80-83.

Yazdanpanah, F., Alvarez-Martinez, C., & Jimenez-Gonzalez, D. (2014). Hybrid dataflow / von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems*, *25*(6), 1489-1509.

Zarate, C., Garfinkel, S., Heffernan, A., Gorak, K., & Horras, S. (2014). A survey of XOR as a digital obfuscation technique in a corpus of real data. Retrieved from https://calhoun.nps.edu/bitstream/handle/10945/38680/NPS-CS-13-005.pdf?sequence=1&isAllowed=y