Analysis and Application of Various Machine-Level Obfuscation Techniques

Brandon Richardson
Software Engineering
University of New Brunswick
Fredericton, New Brunswick
brandon1024.br@unb.ca

Alex Amos
Computer Science
University of New Brunswick
Fredericton, New Brunswick
aamos3@unb.ca

***Abstract*—The concept of software obfuscation is important and widely used in the software industry. Developers and cyber criminals often employ obfuscation techniques to challenge attempts to reverse engineer software, but these techniques are also applied to protect intellectual property, to reduce network traffic congestion, or sometimes to maintain security through obscurity. This paper presents the three classifications of software obfuscation and explores various different techniques that are often used to conceal or obscure an algorithm. We analyse the benefits, limitations, and application of different obfuscation techniques.**

***Index Terms*—obfuscation, de-obfuscation, reverse engineering, malware, disassembly, security**

I. Introduction

In 2017, one of the most devastating ransomware attacks, infamously known as WannaCry, propagated to computers around the globe. Organizations and individuals affected by the attack saw their files encrypted and held at ransom. The attackers demanded an amount in Bitcoin be transferred to an account before the victim might regain access to their files [1].

Given its scale and worldwide implication, the attack was quickly met with significant response from security experts and researchers. The malware was reverse engineered and inspected with the aim of thwarting further spread of the ransomware. Analysis of the malware led to the discovery of a kill switch that allowed analysts to temporarily stop further spread of the malware [2]. This granted time to implement defensive measures and patch systems that were vulnerable to the attack.

Software obfuscation techniques are often used in malware to avoid detection or to hinder efforts to reverse engineering the software. The complexity of obfuscation techniques is constantly evolving, resulting in malware that is increasingly difficult to reverse engineer. In the case of WannaCry, the attack likely would have been far more devastating had analysts been unable to reverse engineer the kill switch. Hence it is important to understand current techniques and follow the trend of innovation lest the next ransomware attack result in more devastating consequences.

Software obfuscation can take many forms but their use is most often legitimate, rather than nefarious. Code minifiers represent the most popular use case for obfuscation. These applications transform source code into a smaller format to be transmitted over a medium restricted by transfer rate. Regardless of their intended purpose, all obfuscations must one simple requirement: the obfuscated program must maintain the same functionality as the original source program and must have a similar run-time and digital footprint. Security-focused obfuscations must also ensure it be considerably more difficult to re-engineer the obfuscated program versus the source program.

In this paper, we investigate common obfuscation techniques to achieve *security through obscurity*. The techniques presented in this paper target static program analysis and program comprehension, or some combination thereof. We begin by describing techniques for obscuring data and information in section III. Then in section IV, we describe techniques for obscuring control flow. Finally, in section V we describe strong hybrid techniques that utilize elements of both types of obfuscation and demonstrate a motivating example.

II. Literature Review

The success of obscuring a specific detail of–or the entire implementation of–an algorithm is measurable in terms of (a) its resistance to disassembly, and (b) its resistance to static analysis. Modern analysis tools are quite powerful, and circumvention of these tools requires deliberate decisions in the design of obfuscated software.

A. Thwarting Disassembly

Disassembly is the process of translating machine instructions from an executable or object file to human-readable assembly instructions. By extension, a disassembler–the program that performs the disassembly–is a reverse-compiler of sorts.

One of the primary goals of software obfuscation is to inhibit ones ability to reconstruct the high-level structure of a program [5]. This can be achieved by frustrating the algorithms that perform the disassembly. This was centre to the research by Cullen Linn and Saumya Debra from the University of Arizona who demonstrated various techniques that can be employed to exploit the nature of those algorithms.

1) Frustrating a Linear Sweep Disassembly Algorithm: A linear sweep disassembler works by inspecting an executable and disassembling each instruction as it is encountered in a sequential manner. This type of disassembler is particularly

susceptible to errors when junk bytes are intentionally inserted that appear to form a valid machine instruction. It was found that this method was able to obtain a *confusion factor* of between 15% and 42% on average (which correlates to the number instructions would be incorrectly interpreted by the disassembler) through techniques such as *branch flipping* and *call conversion* [5].

2) Deceiving a Recursive Disassembly Algorithm: Recursive disassembly algorithms work by following the control flow of the input program. This type of algorithm is strong since it can simply *jump around* junk bytes and jump tables by following the flow of the algorithm. However, recursive disassembly algorithms can sometimes fail to disassemble reachable code if the algorithm is unable to determine all possible targets of a jump. One technique described in the research by Linn and Debra was the use of *branch functions*: a function that returns the address to a branch target [5]. A strong implementation of such a function could accept as arguments an offset and an address and produce a new address a new address whereby the disassembler would need to evaluate the operation to successfully compute the branch target [5].

B. Hindering Intraprocedural and Interprocedural Analysis

The task of reverse engineering an algorithm is tremendously difficult. Intelligent automated tooling that perform static program analysis can be effective in reverse engineering, but its effectiveness is subject to certain limitations. Static analysis must be supplemented by human program comprehension. However, humans are decidedly ill-equipped to process highly complex systems, and so obfuscation that targets static analysis can in turn greatly increase time required to reverse engineer a program.

Interprocedural static analysis is a rather difficult to accomplish in practice, so applying obfuscation techniques to confuse or hinder such analysis can be quite effective, according to Toshio Ogiso, Yushuke Sakabe, Masakazu Soshi et al. from the Japan Advanced Institute of Science and Technology [4]. In their research published in the journal *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, they proved and demonstrated that the problem of identifying the address of a function in the presence of arrays of function pointers is NP-hard [4].

1) Use of Function Pointers: Utilizing arrays of function addresses can be an effective approach to confusing static analysis and program comprehension. In their research, Ogiso et al. devised an obfuscation technique that involved decomposing functions into smaller procedures (function outlining), constructing an array of function addresses and using the array of addresses to invoke those procedures. This technique was said to result in an exponential increase in complexity with linear increase of program size [4].

2) Creating Unrealized Code Paths: Like recursive disassembly algorithms mentioned earlier, static analysis tools follow the control flow of a program. Increasing the number of code paths in a program will frustrate static analysis, resulting in a more obscure implementation. One approach to achieve this is to introduce an intermediary between a calling function and a called function. In order to invoke a given function, the caller invoke a mediator that switches on a given argument and returns the result of the appropriate function [4]. This results in a more complex web of control flow that makes programs more difficult to comprehend.

III. Data-Based Obfuscation

Attackers sometimes embed a kill switch into the malware they deploy. WannaCry was fitted with an interesting type of kill switch: when it installs itself through the *dropper*, the malware attempts to establish an outbound connection to the domain *http://www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com* hard-coded in the executable. If successful, the malware terminates itself. If the domain is registered and a connection can be made to it, the malware will be prevented [1].

Even the most simple applications need to manage their data, and occasionally, application-critical data needs to be obscured to make extracting that data as difficult as possible. In cases such as the WannaCry kill switch, concealing data effectively is critical. This is where data-based obfuscation could be made useful. There are many uses for data-based obfuscation and each type will be examined in detail below.

A. Constant Unfolding

Modern compilers are able to perform various optimizations during the compilation process. One of the most basic compiler optimizations is known as *constant folding*, where the result of an arithmetic operation is performed at compile-time, rather than run-time, thereby *folding* the expression into a constant. This optimization is itself a simple form of obfuscation.

A simple of example of this can be demonstrated with the C application shown in figure 1.

```
#include <stdio.h>

int main(void)
{
    int a = 14 * 6;
    printf("%d", a);
    return 0;
}
```

Fig. 1. When compiled with a modern compiler such as the GNU C Compiler (GCC), the expression `14 * 6` will be folded to the value `84`.

A disassembly `main()` is shown in figure 2. The instruction `movl $0x54,-0x4(%rbp)` shows the result of constant folding of the expression `14 * 6` to `0x54`.

Constant unfolding is a technique that effectively does the opposite–replacing a constant or expression with a more complex computation, thereby obscuring the effective value.

Consider the example shown in figure 3 which doubles the integer value in the variable `a`.

```
Dump of assembler code for function main:
 0x0000113a <+1>:    mov     %rsp,%rbp
 0x0000113d <+4>:    sub     $0x10,%rsp
 0x00001139 <+0>:    push    %rbp
 0x00001141 <+8>:    movl    $0x54,-0x4(%rbp)
 0x00001148 <+15>:   mov     -0x4(%rbp),%eax
 0x0000114b <+18>:   mov     %eax,%esi
 0x0000114d <+20>:   lea     0xeb0(%rip),%rdi
 0x00001154 <+27>:   mov     $0x0,%eax
 0x00001159 <+32>:   callq   0x1030 <printf@plt>
 0x0000115e <+37>:   nop
 0x0000115f <+38>:   leaveq
 0x00001160 <+39>:   retq
End of assembler dump.
```

Fig. 2. A disassembly of the `main()` from the example in figure 1 shows that the expression `14 * 6` was be folded to the value `84`.

```
#include <stdio.h>

int main(void)
{
    int a = 116;
    int b = (((a + 0x8f63) << 3) - (0x47b18)) >> 2;
    printf("%d", b);
    return 0;
}
```

Fig. 3. A more complex example that demonstrates constant unfolding. Here, the value `a * 2` is computed through an obscure computation that utilizes binary arithmetic and constants.

A disassembly of `main()` shown in figure 3 will look quite complex. As seen in figure 4, the operation of doubling an integer is obscured by replacing the simple expression with a more complex computation involving binary operations and arithmetic with seemingly irrelevant constants.

This form of obfuscation can be quite effective in hindering program comprehension, especially with the use of more complex operations or by utilizing seldom used machine instructions. In most cases, there will be a trade-off between performance and obscurity–more complex constant unfolding results in more instructions and lesser performance.

The example from figure 3 is less resistant to static analysis since the expression translates to a sequence of linear instructions that can be easily evaluated [6]. As we will see in section IV, this technique is most effective when combined with complex intermediate control structures and external input (*opaque predicates*, in particular).

B. Data-Encoding Schemes

When inspecting the binary data of an executable or malware artifact, textual data encoded in the binary can be easily read if not obscured properly. In the context of the WannaCry kill switch, the hard-coded domain would need to be encoded in some other format such that the string may not be easily found.

Suppose a naive malware implementation stored a special string (perhaps an encryption key, url, or any other important data string) in a statically-allocated buffer. A simple example is shown in figure 5. Simply inspecting the content of the

```
Dump of assembler code for function main:
 0x00001139 <+0>:    push    %rbp
 0x0000113a <+1>:    mov     %rsp,%rbp
 0x0000113d <+4>:    sub     $0x10,%rsp
 0x00001141 <+8>:    movl    $0x73,-0x8(%rbp)
 0x00001148 <+15>:   mov     -0x8(%rbp),%eax
 0x0000114b <+18>:   add     $0x8f63,%eax
 0x00001150 <+23>:   shl     $0x3,%eax
 0x00001153 <+26>:   sub     $0x47b18,%eax
 0x00001158 <+31>:   sar     $0x2,%eax
 0x0000115b <+34>:   mov     %eax,-0x4(%rbp)
 0x0000115e <+37>:   mov     -0x4(%rbp),%eax
 0x00001161 <+40>:   mov     %eax,%esi
 0x00001163 <+42>:   lea     0xe9a(%rip),%rdi
 0x0000116a <+49>:   mov     $0x0,%eax
 0x0000116f <+54>:   callq   0x1030 <printf@plt>
 0x00001174 <+59>:   nop
 0x00001175 <+60>:   leaveq
 0x00001176 <+61>:   retq
End of assembler dump.
```

Fig. 4. A disassembly of the example from figure 3 shows the extent of the obscurity of computing the value in the variable b.

executable binary will reveal the string. A portion of the compiled binary is shown in figure 6, revealing the secret string.

```
#include <stdio.h>

static char secret[] = "some secret";

int main(void)
{
        printf("%s", secret);
        return 0;
}
```

Fig. 5. In this example, a secret is encoded in a statically-allocated string buffer. This string will be easily visible when inspecting the executable binary.

```
.=.....6.......
........(@.....some secret.GCC:
 (GNU) 9.2.0.,.............9..
.............................
................?...9.........
```

Fig. 6. The secret string from the example in figure 5 can be seen when inspecting the binary.

To obscure the string, we can employ one or more encoding schemes. The simplest data-encoding scheme most commonly used in malware is the *XOR cipher*. This cipher relies on the *reversible* property of an exclusive disjunction; a ciphertext obtained by bitwise exclusive disjunction of a key and plaintext can be reversed by performing the same operation with the key and the ciphertext. More formally,

$$p = (p \oplus k) \oplus k \tag{1}$$

We can apply this concept to the secret string from figure 5 with a repeating 8-bit key to conceal the string. Further, we

will make use of the constant unfolding technique to obscure the key. The result is shown in figure 7.

```
#include <stdio.h>

static char secret[] = {
    0x9b, 0x87, 0x85, 0x8d, 0xc8, 0x9b,
    0x8d, 0x8b, 0x9a, 0x8d, 0x9c, 0xe8,
    0xfa, 0x71, 0x00, 0x12
};

int main(void)
{
    char a = 116;
    char b = (((a + 0x8f63) << 3) - (0x47b18)) >> 2;
    for (char *p = secret; (char)(a << 1) ^ *p; p++)
        printf("%c", (*p) ^ b);

    return 0;
}
```

Fig. 7. Combining constant unfolding and data-encoding schemes truly obscures the operation. Unreachable garbage characters are appended to the secret to mask the true length of the string. This code sample is effectively equivalent to the sample shown in figure 5.

Cryptographic algorithms may also be used as data-encoding schemes. Full or naive implementations of seldom used or deprecated cryptographic algorithms, like 160-bit ECDSA or 80/112-bit 2TDEA, can provide adequate complexity when combined with other obfuscation techniques.

Keep in mind that security and obscurity are two different concepts. In most cases, encoded data need not be cryptographically secure, merely hidden behind some complicated operation. Cryptographic algorithms are characteristically complex and do provide a level of obscurity when utilized effectively. And so, the particular use of a encoding scheme is dependent on the usage thereof. Using an XOR cipher might be sufficient over using a cryptographic algorithm like AES or DES, but this depends on the local complexity.

C. Arithmetic Substitution

Much of the software we build today is used to carry out some arbitrary task. Whether it be to control a smart thermostat or to distribute the energy demand on a power grid, our software executes a sequence of mathematical operations to manipulate data. These individual mathematical instructions are based on fundamental rules and identities. Consider how numbers are represented in modern computers in two's complement. In this scheme, we can subtract two numbers by manipulating the subtrahend and performing a simple binary addition. This can be represented using the following relationship:

$$m - s = m + (\sim s + 1) \tag{2}$$

Modern computers are able to carry out subtraction in two's complement in hardware; a subtraction instruction `sub` does the work of binary negation and the addition of 1. Accordingly, compilers will utilize those instructions to reduce program runtime.

Arithmetic substitution obfuscation increases the complexity of a program by replacing an expression with an identical expression based on some fundamental identity. In our previous example, this might mean to replace subtraction instructions with binary negation and addition.

Of course, this is not limited to identities and axioms of mathematics. We can apply logical identities and axioms of discrete mathematics, theorems from number theory, and a variety of other techniques determined on a case-by-case basis.

One particularly strong example of this applies one of the most famous theorems in number theory, known as Fermat's little theorem. The theorem states that for some prime number p and some integer a, the number $a^p - a$ is an integer multiple of p. Expressed mathematically,

$$a^p \equiv a \pmod{p} \tag{3}$$

Likewise, this can be expressed as follows.

$$a^{p-1} \equiv 1 \pmod{p} \tag{4}$$

This theorem is particularly interesting given that it will always evaluate to the value 1 regardless of the prime number p and integer a chosen.

To demonstrate this theorem applied to obscure some computation, consider the example in figure 8 which computes the value 1024.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
        int r = 1, e = 12, a = argc;
        do {
                if (e & 1)
                        r *= a;

                e >>= 1;
                if (e)
                        a *= a;
        } while (e);

        int len = (r % 13) * 617 + 407;
        printf("%d\n", len);

        return 0;
}
```

Fig. 8. Computation of the value 1024, obscured using arithmetic substitution using Fermat's little theorem. Exponentiation by squaring is used to calculate the power. The integer a is taken from the number of arguments supplied to the program.

Note the interesting property that the value of the base a is insignificant–any value thereof will yield the same final result of 1 (with a caveat, described later). This example makes use of the number of arguments (`argc`) as the value of a, which will frustrate program comprehension and greatly inhibit static analysis.

This method works well up to a certain point and breaks down when a^{p-1} cannot be represented within fixed-width

integer. In the example, this is occurs when $a = 5$. This can be improved by applying memory-efficient modular exponentiation techniques. When combined with large integer arithmetic (GNU GMP library can be used and is licensed under the LGPL v3) and larger prime numbers, this technique can be quite effective in obscuring an algorithm implementation. An example of such an implementation can be found in figure 9.

```
#include <stdio.h>
#include <gmp.h>

int main(int argc, char *argv[])
{
        mpz_t a, p, e, r;

        mpz_init_set_str(a, "96988", 10);
        mpz_init_set_str(p, "A72754B749C1", 13);
        mpz_init_set(e, p);
        mpz_sub_ui(e, e, 1);

        mpz_init(r);
        mpz_powm(r, a, e, p);

        mpz_out_str(stdout, 10, r);
        mpz_mul_ui(r, r, 617);
        mpz_add_ui(r, r, 407);

        mpz_clears(a, p, e, r, NULL);
        return 0;
}
```

Fig. 9. Obfuscating the computation of the value 1024 by arithmetic subtitution of Fermat's little theorem using arbitrary precision arithmetic provided by the GMP library.

IV. Control-Based Obfuscation

Analysts can often rely on the predictability of compilers when translating control flow constructs in unobfuscated code. Compilers frequently break down algorithms into smaller structures that follow common patterns, and assumptions can be made that allow analysts to establish a high-level understanding of the control flow of a program [7]. For instance,

- a `call` instruction will usually indicate a function invocation,
- a `ret` instruction can be used to identify function boundaries,
- in a conditional branch both sides can feasibly be taken,
- in a compiled program, all code in a block will be sequentially located.

Control-based obfuscation attack these common patterns and assumptions by effectively disrupting the predictable nature of compiler output. It achieves confusion through the use of complicated control structures that render the cost of analysis and reverse engineering prohibitively high.

There are multiple techniques of control-based obfuscation that will be broken down throughout this section.

A. Function Inlining and Outlining

1) Function Inlining: Function inlining is the process of embedding the code of a subroutine into the calling function at the location where the subroutine is invoked [7]. Rather than performing a jump to that routine, execution of the routine will be carried out within the calling function. This has the overall effect of flattening the call graph.

Experienced C developers might recall that the standard C language offers function inlining capabilities through the use of the `inline` keyword. Functions qualified with the `inline` compiler directive may not be inlined however; this keyword serves only as a hint to the compiler but might not be realized during compilation. GCC and Clang both support function attributes to force inline expansion of functions. Figure 10 demonstrates the usage thereof.

```
__attribute__((always_inline))
static inline void array_shift(char *argv[],
                int arg_index, int *len, int count)
{
        if (*len <= 0 || arg_index >= *len)
                return;

        int new_len = *len - count;
        for (int i = arg_index; i < new_len; i++)
                argv[i] = argv[i + count];

        *len = new_len;
        argv[new_len] = NULL;
}

int parse_options(int argc, char *argv[],
                struct command_option options[],
                int skip_first, int stop_on_unknown)
{
        int new_len = argc;
        if (argc < 0)
                BUG("negative argv length");

        if (skip_first)
                array_shift(argv, 0, &new_len, 1);

...
```

Fig. 10. The `always_inline` function attribute can be used to force inline expansion of functions.

This technique has the effect of significantly increasing the complexity of function bodies, rendering program comprehension hopeless. The primary drawback with this technique is that frequently-used inline functions can dramatically increase the overall size of the executable.

2) Function Outlining: Function outlining is effectively the reverse of inlining. Rather than embedding a subroutine into the calling function, arbitrary *chunks* of code in the calling function will be extracted into independent routines that will be invoked from the calling function [7]. Unlike function inlining, this technique has the effect of increasing the depth of the call graph.

This technique introduces complicated branching structures that challenges static analysis and program comprehension. Logical chunks of code are severed by complicated function calls, thereby increasing cognitive complexity. This technique comes at the cost of increased branching overhead, which can be significant in systems with hardware instruction pipelining.

B. Destruction of Locality

When describing the *principle of locality* in software, *sequential locality* can be used to describe the predictable placement of code within an executable. Instructions are typically arranged in recognizable patterns, executed sequentially until returning from the function. Compiled binaries have a tendency to exhibit *spatial locality*, meaning that nearby memory locations are more likely to be referenced in the near future than distant ones [7]. This is often desired to improve the performance of systems with memory caching (resulting in fewer page faults), and advanced branch prediction in systems with hardware instruction pipelining.

Destruction of locality is just that, the destruction of spatial locality resulting in an executable with a control flow that is far more complex. This technique involves the creation of unconditional branches within the code that will cause the control flow to jump to different (often distant) instruction locations. This attacks program comprehension and intraprocedural analysis.

Figure 11 demonstrates this concept. This function adds 3 numbers and moves the answer to `RESULT`. By itself this example is quite simple, but the use of unconditional `jmp`'s causes the control flow to jump to different locations in the instruction sequence. This causes sequential instructions to be intertwined together effectively breaking sequential locality.

```
Dump of assembler code for function main:
 0x00000000:  jmp_1:
 0x00000001:       add     %al,NUM2
 0x00000002:       jmp     jmp_2
 0x00000003:  start:
 0x00000004:       mov     %al,NUM1
 0x00000005:       jmp     jmp_1
 0x00000006:  jmp_3:
 0x00000007:       mov     RESULT,%al
 0x00000008:       jmp     jmp_4
 0x00000009:  jmp_2:
 0x0000000a:       add     %al,NUM3
 0x0000000b:       jmp     jmp_3
 0x0000000c:  jmp_4:
 0x0000000d:       NOP
End of assembler dump.
```

Fig. 11. An assembly program that adds 3 numbers and uses unconditional `jmp` instructions to break sequential locality upon compilation.

It should be noted that with the proper use of a control flow graph and using dynamic analysis to understand the result of running this function based on selected inputs can nullify the use of this obfuscation method. However, when used in more complex functions and combined with other obfuscation methods, this can sufficiently hinder disassembly, and human readability of disassembled code.

C. Processor-Based Control Indirection

In the beginning of section IV, we outlined that compilers often behave in a predictable manner with respect to control structures. We indicated that a `ret` instruction typically hints to the existence of a function in that particular instruction locality. Processor-based control indirection disrupts this predictability by effectively masking `jmp` instructions with a sequence of instructions that performs the same operation [7].

To achieve this effect, a `jmp` instruction can be replaced with a combination of `push` and `ret` instructions.

```
Dump of assembler code for function main:
 0x00000000:  push    target_addr
 0x00000001:  ret
End of assembler dump.
```

Fig. 12. Using a combination of `push` and `ret` instructions, we achieve a similar effect to a `jmp` instruction.

This will frustrate efforts to decompile the program since the sequence of instructions would usually indicate that a subroutine boundary exists there, while instead these instructions belong to the inner body of a subroutine. In other words, a decompiler might mark these instructions as the end of a function body, when in reality these instructions may simply branch to a different location within the same routine.

Similarly, we can replace `jmp` instructions with `call` instructions to the same effect [7]. Together with the previous example, we can construct spurious functions with the explicit aim to confuse, thereby furthering the disruption to inter- and intra-procedural analysis. This is demonstrated in figure 13.

```
Dump of assembler code for function main:
 0x00000000:  call    target_addr
 0x00000001:  <Junk Code>
 ....
 0x00000020:  target_addr
 0x00000021:  <continuing function>
End of assembler dump.
```

Fig. 13. In this function a disassembler will mark `0x00000020` as a new function and wait for a return to address `0x00000001`, which will never come.

D. Function Pointers

Many methods of control flow obfuscation operate by attempting to bypass a specific form of functionality like processor indirection, or create many displacements to confuse disassemblers such as in locality destruction and function outlining. Function pointers is more unique. This technique abuses function references to hinder the process of identifying the address of a function call.

The function pointers technique is an extremely effective method of obfuscation that can protect against call graph generation and interprocedural analysis. The basis of this technique is the use of arrays of function pointers and complex mathematical expressions to perform function calls instead of calling a function directly by address. This will be demonstrated through a series of examples. These examples are being referenced from *Software Obfuscation on a Theoretical Basis and Its Implementation*, the research from which this method was created [4].

Figure 15 demonstrates a simple program that accepts two numbers and prints out $a + b$ or $a - b$. Through a series of steps this program will be broken down and obfuscated using function pointers.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int a, b;
        scanf('%d', &a);
        scanf('%d', &b);

        if (a == b) {
            printf('%d', a+b);
        } else {
            printf('%d', a-b);
        }

        return 0;
}
```

Fig. 14. Function pointers step 0, setup.

In figure 15, a few subroutines are extracted from the main function and called in place of the original instructions. Function outlining is used to perform the subroutine extraction. However, if there are pre-exisitng subroutines being called, this step can be skipped and those functions used instead. It should also be noted that in some cases similar to this example, some variables may need to be changed or made global so that they can be accessed by all functions.

```
#include <stdio.h>
#include <stdlib.h>
int a, b;
void func1(){printf('%d', a+b);}
void func2(){printf('%d', a-b);}
int main(void)
{
        scanf('%d', &a);
        scanf('%d', &b);

        if (a == b) {
            func1();
        } else {
            func2();
        }

        return 0;
}
```

Fig. 15. Function pointers step 1, extracting subfunctions.

In figure 16 a function pointer `fp` is created that is used to call each subroutine. Two new if-else statements are created that use complex mathematical expressions to determine which function `fp` points to. Upon further inspection, each statement has only one realizable code path. The first one bit shifts by 1, meaning the value is multiplied by two. As such, modulus division by 2 will always result in 0. In the second statement, the three part multiplication will always be divisible by 6, also resulting in 0. Therefore, the first statement will always be true and the second, false. These expressions are very difficult to resolve after compilation, making it extremely difficult to determine the value of `fp`. Furthermore, these expressions are protected from arithmetic folding since variables are involved.

```
#include <stdio.h>
#include <stdlib.h>
int a, b;
int (*fp)();

void func1(){printf('%d', a+b);}
void func2(){printf('%d', a-b);}
int main(void)
{
        scanf('%d', &a);
        scanf('%d', &b);

        if (a == b) {
            if (((a+3) << 1) % 2 == 0){
                fp = func1;
            }
            else{
                fp = func2;
            }
            (fp)();
         }
         else {
            if ((b-2) * (b-1) * b % 6 != 0){
                fp = func1;
            }
            else{
                fp = func2;
            }
            (fp)();
        }

        return 0;
}
```

Fig. 16. Function pointers step 2, making use of function pointers.

In figure 17 the example is obfuscated further by using an array to store the function pointers. Another function is created, `func0` that is used for array displacement; it uses the variable `a` to return an even number regardless of the variable's value.

Each function is assigned to an array location. The first few locations are purposely assigned, specifically, `A[0]` to `A[3]`. After that, each function is randomly assigned to different locations. These randomly assigned locations add further confusion and create new unrealizable code paths. Next, `fp` is assigned an array location using `func0`. Since, func0 returns an even number thus being divisible by 2, the expression will result in 0, assigning `func0` to `fp`. Now, `fp` is called to reassign itself a new array location for each subroutine call. In the first if-else statement, `A[2]`, `func1` is assigned, and `A[3]` for the second statement which is `func2`. Before the second if-else statement `fp` is assigned the location `A[b & 1]`. This simply assigns `fp` to location 0 or 1 which will reset it back to `func0`.

In each step of the example it can be seen that the program becomes increasingly more complex. However, with each step of obfuscation the program maintains the same functionality.

```c
#include <stdio.h>
#include <stdlib.h>
int a, b;
int (*fp)();
int (*A[10]) ();
int func0() { return ((a-1) * a; }
void func1(){printf('%d', a+b);}
void func2(){printf('%d', a-b);}
int main(void)
{
    A[0] = A[1] = func0;
        A[2] = A[9] = func1;
        A[3] = func2;
        A[4] = A[6] = func0;
        A[5] = A[7] = A[8] = func2;
        scanf('%d', &a);
        scanf('%d', &b);

        fp = A[(func0() % 2) * a * b];
        if (a == b) {
            if (((a+3) << 1) % 2 == 0){
                fp = A[((fp) () % 2) + 2];
            }
            else{
                fp = A[((fp) () % 2) + 4];
            }
            (fp)();
         }
         else {
            fp = A[b&1];
            if ((b-2) * (b-1) * b % 6 != 0){
                fp = A[((fp) () % 2) + 7];
            }
            else{
                fp = A[((fp) () % 2) + 3];
            }
            (fp)();
        }

        return 0;
}
```

Fig. 17. Function pointers step 3, moving pointers to an array.

Due to the difficulty of determining the output of each mathematical expression and the use of randomly placed array elements, the problem of determining address of the function pointer has been shown to be NP-hard [4]. In comparison to other algorithms with much higher complexities such as N^2, 2^N, etc, this may seem too simple for a skilled reverse engineer. However, interprocedural analysis is fundamentally an extremely difficult process. As such, adding any increased difficulty, especially an increased NP complexity to this process makes it near impossible to complete. The main purpose of obfuscation is to make it easier to build a new functionally similar program from scratch than to reverse engineer the already built program. The use of function pointers undeniably accomplishes this goal.

V. Hybrid Obfuscation Techniques

Among all the different obfuscation techniques discussed in this paper, virtual machine obfuscation is arguably the strongest class of techniques to obscure algorithm implementations. The essence of this technique is the implementation of a bytecode interpreter that executes bytecode instructions, both of which are embedded within the obfuscated program.

The details of the bytecode instruction format is left to the discretion of the developer, but often the format is similar to machine instructions (`add`, `sub`, `jmp`, `hlt`, etc.). This gives some flexibility in determining which instructions are most important and allows the developer to combine common operations into a single instruction (reading a byte from standard input, and putting the status of the operation into a given register, for example).

The most difficult part to this technique is to implement the (obfuscated) algorithm in bytecode. Depending on the bytecode instruction set chosen, this process is likely going to be similar to writing instructions in assembly language. Alternatively, one might develop a step in the compilation process that compiles selected code from a higher-level language (C) into machine instructions for a target CPU architecture, then transforming these machine instructions into bytecode. This would involve specialized tooling which could be laboursome to develop.

In the context of malware–computer worms in particular– this technique is quite potent because of it's ability to take advantage self-mutating bytecode. Before spreading to other computers on the network, the obfuscated malware might permute it's bytecode instruction format or instruction set by some random value. Malware of this type are said to be *polymorphic*, and this class of malware is much more difficult to reverse engineer, which hinders efforts to stop the malware from spreading. This is akin to how viruses mutate and spread in human or animal populations resulting in significant difficulty in developing an effective vaccine.

Of course, this form of obfuscation is particularly prone to sizeable overhead which can significantly slow execution. Even the most optimized VM implementations will have some overhead. Therefore it's usually not feasible to obfuscate an entire algorithm. Instead the most critical/vulnerable parts of the algorithm are implemented in bytecode and the other details are implemented in the source language.

To demonstrate this technique, we built a simple virtual machine and bytecode instruction set to implement an obfuscated implementation of a base64 encoder. This this section, we provide implementation details and in the next section V-B we benchmark this program to compare the performance of the obfuscated program versus the `base64` tool available under GNU Coreutils. The full implementation can be found in our public GitHub repository, a link to which can be found in appendix A.

A. Implementation of a Simple VM to Obfuscate an Algorithm

To implement a virtual machine, one must first establish an instruction set and instruction format. In our implementation we opted for a minimal instruction set comprised of 23 unique op codes. The instruction set has 12 arithmetic and logic instructions, 4 memory instructions, 4 control flow instructions and two IO instructions. These instructions are listed in figure 18.

```
#define INST_READ    0x00
#define INST_WRITE   0x01
#define INST_SET     0x02
#define INST_ADD     0x03
#define INST_ADDI    0x04
#define INST_SUB     0x05
#define INST_SUBI    0x06
#define INST_SHL     0x07
#define INST_SHR     0x08
#define INST_ANDI    0x09
#define INST_ORI     0x0a
#define INST_XORI    0x0b
#define INST_AND     0x0c
#define INST_OR      0x0d
#define INST_XOR     0x0e
#define INST_ST      0x0f
#define INST_LD      0x10
#define INST_STR     0x11
#define INST_LDR     0x12
#define INST_JMP     0x13
#define INST_JZ      0x14
#define INST_JNZ     0x15
#define INST_HLT     0x16
```

Fig. 18. Bytecode instruction set for our obfuscated VM implementation of a base64 encoder.

The virtual machine implementation supports 16 general-purpose unsigned integer registers and 256 bytes of memory, although only 7 registers and 67 bytes of this memory is utilized by our bytecode program.

Instructions are comprised of a 8-bit op code, three 4-bit register operand addresses, and an 8-bit immediate value field.

```
struct instruction {
        unsigned op: 4;
        unsigned reg1: 4;
        unsigned reg2: 4;
        unsigned reg3: 4;
        unsigned imm: 8;
};
```

Fig. 19. Instruction format structure.

Instructions are embedded into the program as an array of `instruction` structures. The format of the program is similar to that of assembly language. Preprocessor directives are used to allow the program to be written in a more human-friendly format. A fragment of the bytecode program is presented in figure 20.

Implementation of the bytecode interpreter is rather simple. It first allocates space for general-purpose registers and memory, then executes instructions until the INST_HTL instruction is encountered. The instruction executor is responsible for executing individual instructions and changing the state of the virtual machine (program counter, for instance).

Our implementation utilizes the function pointers technique described in section IV-D to obscure the virtual machine implementation. This was achieved using the constructor and destructor function attributes to allocate and initialize an array of function pointers upon entering `main()` and releasing those resources after `exit()`. Each bytecode instruction

```
...
// beginning of byte_read routine
/* 0003 */ {INST_READ, REG(4), REG(0)},
/* 0004 */ {INST_JNZ, REG(0), REGX, REGX, 0x06},
/* 0005 */ {INST_JMP, REGX, REGX, REGX, 0x0a},
/* 0006 */ {INST_STR, REG(4), REG(2)},
/* 0007 */ {INST_ADDI, REG(2), REG(2), REGX, 1},
/* 0008 */ {INST_SUBI, REG(1), REG(1), REGX, 1},
/* 0009 */ {INST_JNZ, REG(1), REGX, REGX, 0x03},

// if no bytes were read, jump to end of program
/* 000a */ {INST_JZ, REG(2), REGX, REGX, 0x3f},
...
```

Fig. 20. A fragment of the base64 encoder implemented in a custom bytecode instruction language.

has it's own executor function, each with the same function signature. The array of function pointers is used to store references to those executor functions.

Executor functions are inserted into the array using a secure random offset (taken from `/dev/urandom`), and this non-deterministic property ensures strong resistance to reverse-engineering and static analysis.

```
int execute(instruction_reg, gp_reg, memory, pc)
        struct instruction *instruction_reg;
        struct reg gp_reg[16];
        unsigned char memory[256];
        int *pc;
{
        size_t fn_index =
                (offset + instruction_reg->op) % 23;
        return instructions[fn_index]
            (instruction_reg, gp_reg, memory, pc);
}
```

Fig. 21. The instruction executor. The random offset and instruction op code are used to select the correct function reference from the array of function pointers constructed during initialization.

Once compiled, the program accepts input from standard input and prints encoded characters to standard output.

```
$ echo 'encoded string' | ./b64-vm
ZW5jb2RlZCBzdHJpbmcK
$ echo 'encoded string' | base64
ZW5jb2RlZCBzdHJpbmcK
```

Fig. 22. Demonstration of the obfuscated base64 encoder.

B. Performance Evaluation

To evaluate the performance of the obfuscated base64 encoder, we used the open-source `perf` performance analyzing tool maintained alongside the Linux kernel. Our implementation was compared against the GNU Coreutils `base64` encoder, and we used an input size of 50000 characters.

Our findings can be found in table I. To summarize, the obfuscated implementation is considerably slower by 216ms averaged over 50 runs, which represents a 26-time increase in execution time. This execution time is largely spent in reading

and writing bytes to and from standard streams, given that our implementation naively consumes a single byte at a time.

TABLE I
RUN-TIME OF THE OBFUSCATED BASE64 ENCODER AGAINST THE GNU COREUTILS IMPLEMENTATION, AVERAGED OVER 50 RUNS WITH AN INPUT SIZE OF 50000 BYTES.

Command	Average Elapsed Time (sec)	Error ($\pm$ sec)
./b64-vm $<$ input	0.224980	0.000942
base64 $<$ input	0.0087071	0.0000382

APPENDIX A
IMPLEMENTATION OF OBFUSCATED BASE64 ENCODER

The full implementation of the obfuscated base64 encoder described in section V can be found on GitHub: https://github.com/brandon1024/obfuscation-paper-2020

REFERENCES

[1] G. S. Hsiao and D. Kao, "The static analysis of WannaCry ransomware," 2018 20th International Conference on Advanced Communication Technology (ICACT), Chuncheon-si Gangwon-do, Korea (South), 2018, pp. 153-158.

[2] MalwareTech, "How to Accidentally Stop a Global Cyber Attacks". Accessed on: Mar. 4, 2020. [Online]. Available: https://www.malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html

[3] T. Ganacharya, "WannaCrypt ransomware worm targets out-of-date systems," Microsoft Defender ATP Research Team. Accessed on: Mar. 4, 2020. [Online]. Available: https://www.microsoft.com/security/blog/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/?source=mmpc

[4] T. Ogiso, Y. Sakabe, M. Soshi, A. Miyaji, "Software Obfuscation on a Theoretical Basis and Its Implementation," IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, E86-A(1): 176-186, 2003-01. Accessed on: Mar. 5 2020. [Online]. Available: https://dspace.jaist.ac.jp/dspace/handle/10119/4427

[5] C. Linn, S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," In Proceedings of the 10th ACM conference on Computer and communications security, pp.https://www.overleaf.com/project/5e35ea775e63e0000161f5d4 290-299. 2003. Accessed on: Mar. 5 2020. [Online]. Available: http://www2.cs.arizona.edu/solar/papers/CCS2003.pdf

[6] A. Moser, C. Kruegel, E. Kirda, "Limits of static analysis for malware detection," In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007) (pp. 421-430). IEEE.

[7] B. Dang, A. Gazet, E. Bachaalany, S. Josse, "Practical Reverse Engineering," Indianapolis, IN: Wiley, 2014.

[8] J. Ge, S. Chaudhuri, A. Tyagi, "Control flow based obfuscation," In Proceedings of the 5th ACM workshop on Digital rights management '05, 2005, pp. 83-92. Accessed on: Mar. 7 2020. [Online] Available: https://doi-org.proxy.hil.unb.ca/10.1145/1102546.1102561