CONFUSE: LLVM-based Code Obfuscation

Chih-Fan Chen,
cc3500@columbia.edu

Theofilos Petsios,
tp2392@columbia.edu

Marios Pomonis,
mp3139@columbia.edu

Adrian Tang,
bt2349@columbia.edu

Abstract

In the past decade, code obfuscation techniques have become increasingly popular due to their wide applications on malware and the numerous violations of intellectual property caused by reverse engineering. In this work, we examine common techniques used for code obfuscation and provide an outline of the design principles of our tool **C**onfuse. **C**onfuse is an LLVM tool which modifies the standard compilation steps to produce an obfuscated binary from C source code.

Keywords: code obfuscation, CFG alteration, hardening, reverse engineering

1 Introduction

The distribution of current software applications is plagued with vast financial losses due to **reverse engineering** [1]. The term refers to the process of discovering the technological principles of a device, object, or system through the analysis of its structure, function, and operation. It involves recovering and making sense of the higher-level semantics of a compiled binary. Once a company develops an application and releases it, it ceases to have a strong control over who accesses their source code and who doesn't. Rivals or adversaries can gain access to this code and use it for their own benefit without investing as much resources as the original authors. State-of-the-art disassemblers (eg. IDA Pro [2]) and decompilers (eg. Hex-Rays Decompiler [3]) enable people to readily reverse-engineer the inner workings of an executable. The effectiveness of the reverse engineers' efforts is dependant on the amount of time and resources they are willing to put into reversing a particular piece of code. With such threat to intellectual property, there is a strong motivation to find effective ways to protect compiled software.

Code obfuscation involves performing a series of transformations on software, converting it to a harder-to-be reverse-engineered form, which maintains the unmodified software's functionality with a reasonable overhead. The primary motivation for code obfuscation is to protect as much as possible any intellectual property, such as algorithms and data structures, inherent in a software application that is being sold and distributed. Raising the bar for reverse-engineering, code obfuscation attempts try to harden, in terms of time and resources, the disassembling attempts performed by automated systems and possible adversaries.

Our project aims to develop a code obfuscating tool which takes C source code as input and produces an obfuscated binary, which will be more resistant to being reversed-engineered in comparison to the respective unobfuscated executable.

2 Related Work

Existing obfuscation methods and tools fall traditionally into one of the following categories: layout, design, data and control obfuscation [4]. **Layout obfuscation** refers to the process of modifying the layout of the software: deleting comments, changing the name of variables, removing debugging information etc. **Design Obfuscation** [5] refers to the effort being made to obscure the design information of the software. Such actions involve slitting and merging of classes etc. **Data obfuscation** [4] techniques involve array and variable splitting, conversion of data to procedures, changes in variables' lifetime etc. **Control Flow Obfuscation** (CFO) [6] techniques use abstract interpretation and opaque predicates to break up the program's control flow. A very effective technique in this category is **parallelization** [4] and **control flow flattening** with history maintenance [7]. CFO can also be achieved by **hiding control flow information in the stack** [8]. Various algorithms [9] [10] used in software containing **self modifying code** utilize **dynamic obfuscation techniques**. Recent works [11] propose a **signals based approach**: in this case the CFO is achieved through signals used for Inter Process Communication (IPC). Control flow instructions are replaced with trap instructions. When a trap instruction is raised, the respective signal triggers a signal handler which invokes the *restoring function* which will transfer the system control to the original target address.

3 LLVM

For the purpose of our project we will use the the LLVM[12] compiler framework with Clang[13] as the front end. LLVM is one the most popular and frequently used compiler frameworks, largely because of the plethora of its supported languages and architectures. It utilizes the three phase design (front end - optimizer - back end) that provides flexibility and modularity to the compiler and uses its own Intermediate Representation (IR) as an internal form of code presentation.

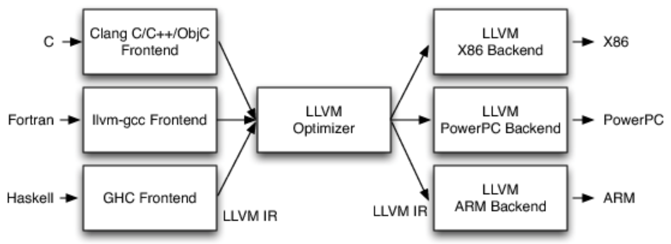


Figure 1: LLVM: Three Phase Design

As presented in figure 1, each front end tests the source code for errors and maps it to LLVM IR. Then a source and architecture independent optimizer makes various transformations on this IR in an effort to make execution more efficient. Finally, depending on the processor family, a suitable back end translates the optimized IR to binary code. Clang is a front end that provides a unified parser for C-based languages and can be used for a variety of purposes such as indexing, source analysis, refactoring etc.

In the following sections we will provide a brief description of the IR properties and of the optimizer's architecture since Confuse will be implemented as part of the optimizer.

3.1 LLVM IR

The aforesaid IR is a first class low-level language, with a small three-address-form virtual instruction set and a potentially infinite number of virtual registers. Unlike common assembly languages it provides some abstraction from the machine internals such as the ABI, even though it is strongly typed and uses labels. The IR can losslessly be transformed to three distinct forms: the textual that can be understood easier by a human, an in memory data structure used for the optimizations and an efficient and dense bitcode binary format. Finally we should note that the IR facilitates the communication between the various compilation phases since it allows a complete representation of the source code without the need to extract information from previous phases thus making each phase completely separate.

3.2 Optimizer Architecture

The optimizer is structured as a series of distinct passes which are built into archive libraries as loosely coupled as possible (i.e. they are intended to be completely independent otherwise they explicitly request analysis information from other passes). Each pass is written in C++, derives the Pass class and produces a modified version of the IR it reads that will be used as input for the next one.

This scheme gives the user the ability to choose which passes she prefers to utilize when optimizing a program by making use of the PassManager. The PassManager makes sure that all necessary passes are included (e.g. all analysis passes needed by an optimization pass) and tries to set the passes in the order that will yield the optimal result.

The most important advantage of this design is that a programmer that wishes to create a new optimization pass can decide the passes that need to be executed before and after the execution of his new pass. This dynamic layout forces minimal overhead to new optimizations since one can use only suitable passes as their prelude and postlude while the not used ones will not be linked to them. Figure 2 shows an example of how a new PassXYZ that does not require all the passes can only be paired with PassA and PassB (and not PassC etc) for efficiency.

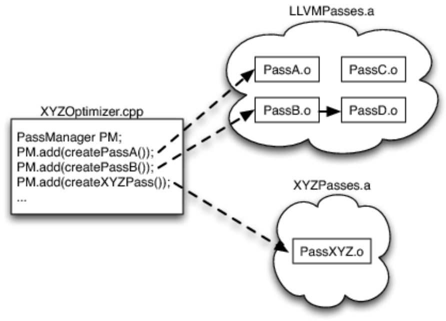


Figure 2: Pass Selection Using PassManager

4 Data Obfuscation

4.1 String Obfuscation

Strings are one of the most expressive data types used in high-level languages and the control flow of many applications relies on the outcome of string comparisons. Let us consider snippet 1:

```
if (strcmp(cmd, "add") == 0)
{
/* handle add */
}
else if (strcmp(cmd, "remove") == 0)
{
/* handle remove */
}
else if (strcmp(cmd, "edit") == 0)
{
/* handle edit */
}
else
{
/* error */
}
```

Code 1: String comparisons

It is trivial for someone to infer that the code of snippet 1 is used for the modification of entities (e.g. files). Comparisons like the one of snippet 1 are very common especially in command-line applications that handle user commands. At the same time they provide valuable insight to the application's programming logic that can lead to a more efficient and accurate reverse-engineering.

Confuse deals with this type of comparisons by making use of cryptographic hash functions. Cryptographic one-way hash functions are functions that map input from a large data set to a bit string in a way that any change to the input would result to a different bit-string and that given the bit-string it is impossible to compute the input. Examples of cryptographic hash functions include SHA-1 [14] and md5[15].

We already know the possible (legitimate) values that the string can take, therefore we can compute their hash values and then at *run-time* use the hash function on the string variable and use its output for the comparisons. The obfuscated version of snippet 1 can be seen in snippet 2:

```
/* known a priori
Hash("add") = h1
Hash("remove") = h2
Hash("edit") = h3
*/

hash = Hash(cmd);

if (hash == h1)
{
/* handle add */
}
else if (hash == h2)
{
/* handle remove */
}
else if (hash == h3)
{
/* handle edit */
}
else
{
/* error */
}
```

Code 2: Obfuscated string comparisons

It is obvious that someone that tries to analyze snippet 2 is unable to deduce its use by looking at the predicates. Moreover in many cases after using this technique we can add more cases and fill their body with junk bytes in order to force the reverse-engineer to waste her time and resources in an effort to infer the behavior of this branch.

4.2 Insertion Of Irrelevant Code

Another technique that is frequently used by obfuscating tools is the insertion of junk code. The purpose of this is to force mistaken conclusions about the control flow of the program by adding unnecessary complexity (e.g. redundant operations) while simultaneously preserving the semantics of the source code. Let us consider snippet 3:

```
int x = 0;
x++;
return x;
```

Code 3: Original Code Sequence

A simplest way is to intersperse actual code with some variables that never been used. This will make the code more complex, forcing the reverse-engineer to waste time and effort sieving out the useless instructions. For example, the irrelevant variable y is introduced to code snippet 4. Since y is not used in the actual computation of x, we can introduce superfluous instructions on y without affecting the original code. However, this kind of junk code is easily found and will probably be eliminate by code optimizer. We must use some other technique to insert junk code.

```
int x = 0, y;
x++;
y = 3;
return x;
```

Code 4: Obfuscated Code Sequence With Irrelevant Variable

Instead of inserting the unused variable, we can use some simple mechanisms to make the sequence significantly less visible. For example, in snippet 5 we added several mathematical operations.

```
int x;
x++;
x = (3 * (x + 4) - 12 ) / 3;
```

```
return x;
```

Code 5: Obfuscated Code Sequence

This transformation is based on the fact that $(b*(t+b)-b*a)/b = (bt+ba-ba)/b = bt/b = t$ and makes the code much less prone to automated reverse-engineering since the disassembler cannot know mathematical identities. Another advantage of this obfuscation is that we do not have to add any new variable into the original program. The two numbers, a and b, is added in the IR level by our obfuscation code. Since a and b are not bounded, they can be used to reproduce any value needed and have no effect to the final result.

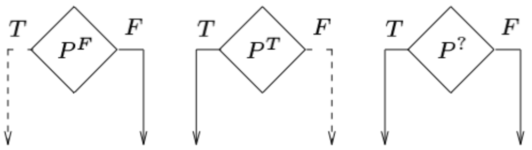
The main disadvantage of the aforementioned techniques is the space/time overhead introduced into the program as these superfluous instructions are executed at run-time. To alleviate this, we can choose to insert irrelevant code as the body of an unconditional branch that is never taken. In this case we refer to this as dead code that will not be executed at run time. The insertion of such dead code is made more effective when used in conjunction with the insertion of opaque predicates. We will describe this in more detail in Section 5.3.

5 Control Flow Obfuscation

5.1 Opaque Predicates & Variables

We implement control flow obfuscation using opaque predicates and variables. In general terms, an opaque variable V has some property q that is known *a priori* to the obfuscator but which is difficult for the deobfuscator to deduce. Similarly, an opaque predicate is a condition that has an outcome that the obfuscator knows *a priori* but which is hard for the deobfuscator to compute, since the static analysis of all branches has exponentially larger cost.

Definition: A **variable V is opaque** if at any point p in a program, V has a property q which is known at obfuscation time. We denote this property by ${V_p}^q$. A **predicate P is opaque** if its outcome is known at obfuscation time. If P always evaluates to *True* at point p, we write ${P_p}^T$, whereas if P always evaluate to *False* we write ${P_p}^F$. When it is uncertain if P will evaluate to true or false, we write ${P_p}^?$[4]:



A predicate is considered *trivial* if a deobfuscator can crack it with local static analysis, that is, by examining a small number of previous commands. A predicate is considered *weak* if a deobfuscator can crack it with global static analysis. The following examples are illustrative of the previous:

```
int x=5;
int y=8;
int v=x/y;
if (v>1) {...}
```

Code 6: Trivial Opaque Predicate

```
int x=5;
<...> //multiple commands where x
    remains unchanged
if (x>1) {...}
```

Code 7: Weak Opaque Predicate

5.2 Choosing the proper predicate

The basic criteria for evaluating the quality of an obfuscating transformation are [16]:

- **Potency**, that is, how much obscurity an obfuscating transformation adds to the program
- **Resilience**, which refers to how difficult it is for an automatic deobfuscator to break the obfuscating transformation
- **Stealth of Secrecy**, which refers to how hard it is to tell if some part of code belongs to the original program or the obfuscation part
- **Overhead**, which corresponds to the differences in the running time of the obfuscated versus the normal program

5.2.1 Predicates based on mathematical identities

An easy way of producing opaque predicates which are always either true or false, is based on number-theoretical identities. The following identities[17] hold for all values of $x, y \in Z$:

- $7y^2 - 1 \neq x^2$
- $3|(x^3 - x)$
- $2|x \vee 8|(x^2 - 1)$
- $\sum_{i=1,\, 2 \not| i}^{2x-1} i = x^2$

The following hold for all values of $x \in N$

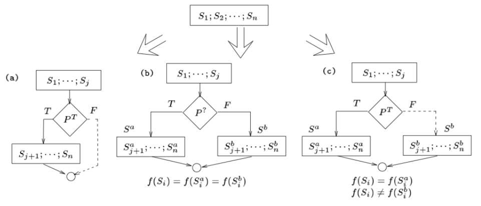
- $14|(3 * 7^{4x+2} + 5 * 4^{2x-1} - 5)$
- $2|\lfloor \frac{x^2}{2} \rfloor$

If we initialize a variable a to a value V_a at some point in the execution of the program, then, say, the predicate $V_a|(x^3 - x)$ will always be true if V_a is 3 and false if V_a is not a multiple of 3. This category of opaque predicates is easy to produce but not very easy to break, though in general they are the strongest of weak predicates, granted that the options for choosing the opaque predicate are limited, and thus, given time and effort, one can bruteforce all possible options at runtime.

5.3 Transformations

5.3.1 Insertion Transformation

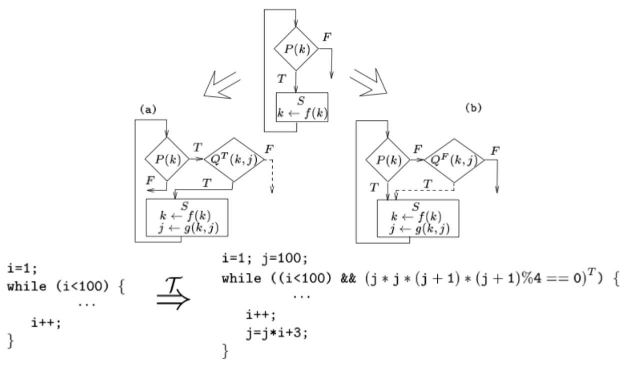
Consider the following figure [4]:



Consider the basic block $S = S_1; S_2; ...S_n$. We insert an opaque predicate P^T into S, splitting it in half (a). The predicate contains irrelevant code since it will always evaluate to True. In figure (b) we split it into two halves and then proceed to create two different obfuscated versions S^α, S^b, using different sets of transformations on each half and we select between them at runtime. Finally in (c) we introduce a bug to S^b and always select S^α.

5.3.2 Loop Condition Insertion Transformation

Consider the following figure [4]:



```
i=1;
while (i<100) {
        ...
   i++;
}
```

```
i=1; j=100;
while ((i<100) && (j*j*(j+1)*(j+1)%4 == 0)^T) {
          ...
   i++;
   j=j*i+3;
}
```

In this example we obfuscate a loop by making the termination condition more complex, adding a predicate that does not modify the number of times the loop gets executed.

The efficiency of the aforesaid methods can improve significantly if we use it in conjunction with the insertion of junk bytes as the code to be executed in the (a priori known) non taken branch of an opaque predicate. A deobfuscator in this case must evaluate both branches (since it cannot predict which one is taken) and therefore this will lead to the ineffective use of time and resources and possibly to mistaken conclusions.

6 Implementation

Our obfuscator will be implemented in C++ since it is fully supported by Clang and will modify C files using the techniques explained above. This decision was based on the fact that C is one of the most popular and frequently used programming languages for applications that are prone to be reverse-engineered and thus its obfuscation would be a valuable contribution to the community.

Confuse will consist of a series of optimization (i.e. obfuscation) passes that will transform the LLVM IR produced by Clang before it is translated to binary code suitable to the processor family.

We note that the modifications applied by these passes depend on the source file given as input. For example, if there is no string comparison in the source code the string transformation pass will not make any change. However, since the opaque predicates mechanism does not rely on such assumptions, **C**onfuse can employ this transformation on any input source code.

7 Testing

While our code obfuscation techniques aim to transform the input program to a form more resistant to reverse-engineering efforts, they must ensure that the original semantics of the program are preserved.

Being very extensive, the LLVM framework has a very robust and comprehensive testing infrastructure. It has two major categories of tests, namely whole programs and regression tests. The whole programs are contained in test-suites which can be compiled using specific compiler flags (to be tested). The output of the compiled programs are then compared to some reference output to verify the correctness of the LLVM functions. The regression tests are small unit tests meant to test specific functionality.

We leverage on the LLVM testing infrastructure to conduct our testing of our obfuscation pass libraries. We design both whole programs and regression tests to be executed with our obfuscation passes to verify that the transformations have not altered original behavior of input programs. The testing is mainly driven by the LLVM Integrated Tester tool, **lit**, that we describe in the next sub-section.

Determining the correctness of a program after program transformation is difficult. To this end, we choose to verify that the semantics of the input program are preserved by comparing the output of a transformed program with a known reference output. We will use this strategy to do incremental testing of our obfuscation functionalities.

7.1 lit - LLVM Integrated Tester

Shipped together with the LLVM framework, **lit** is a portable tool used for executing LLVM and Clang regression and unit tests. It provides a convenient way for developers of LLVM and Clang libraries to easily write and run test files during the development process. **lit** is designed to be a multi-threaded lightweight tool for executing and reporting the success of the regression tests.

7.2 Example Regression Test

A regression test file can be any valid input file (such as C file, bitcode file, LLVM IR file) that is accepted by any LLVM frontend tool. Since our code obfuscation targets C source files, we will design the test files to be C source files, instrumented with **lit**-specific commands for the test verification.

Each test file must contain comment lines starting "*RUN:*" that instruct **lit** how to execute this file. These comment lines contain shell commands that when executed determine if the test is successful or not. **lit** will report a failure if the verification commands return false.

An example of a regression test file is as follows:

```
// RUN: clang -emit-llvm %s -c -o -|
    opt -load lib.dylib -objunk > %t1
// RUN: lli %t1 > %t2
// RUN: diff %t2 %s.out

#include <stdio.h>

void f_div(int i, int *k)
{
  int j = 2;
  *k = i / j;
}

int main()
{
  int k;
  f_div(8, &k);
  printf("%d\n", k);
  return 0;
}
```

Code 8: Simple regression test

This regression test is designed to verify that the following functionalities are preserved after code obfuscation: (1) Simple printing of strings, (2) Integer division, (3) Parameters passed by-value and by-ref. The set of *RUN* commands compiles this C file into a LLVM bitcode file, transforms this bitcode file using our obfuscation pass (specifically junk instructions insertion), executes this obfuscated bitcode file and verifies that the output is *4*. If the output is not what we expect, the diff command will return *false*, and **lit** will report a test failure.

7.3 Test Design

We have structured the test suite to consist of both set of tests specific to the individual obfuscation passes, and those that test the correct functioning combining multiple obfuscation passes together. We detail the tests specific to each obfuscation pass below.

The string obfuscation targets the hiding of strings used in string comparisons. We design the tests by changing the various orders of the parameters we pass into the string compare function. Besides using strings as the actual parameters, we also include tests that pass in *const* string buffers which should be obfuscated by our string obfuscation pass. We also use variants of string comparison functions like *strcmp()* and *strncmp()*.

The junk-code obfuscation involves transforming arithmetic assignment instructions into more convoluted but semantically equivalent instructions. Naturally, we design the tests that make use of arithmetic assignment in a range of situations. This includes using the assignments with different arithmetic operators in *while* loops, *for* loops and plain sequence of arithmetic operations.

The control-flow obfuscation using opaque predicates is much more flexible compared to the above two types of obfuscation as it can practically be used on any kind of code. In view of this, we reuse the tests from the other two types of obfuscations.

8 Results

In this section we will present brief results for every obfuscation technique we implemented in an effort to showcase some of the advantages obtained in the obfuscated file.

8.1 String Transformation

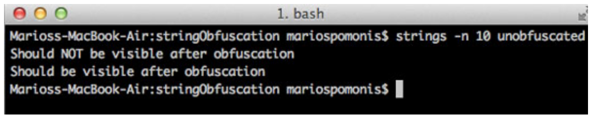
The main advantage of this technique is presented in the respective section, however there is a valuable side effect that can occur in a number of occasions. Let us assume snippet 9:

```
char a[] = "Should␣be␣visible␣after␣
    obfuscation";
const char b[] = "Should␣NOT␣be␣
    visible␣after␣obfuscation";
if (strcmp(a, b) == 0) {
/* code that does not use b */
}
/* code that does not use b */
```

Code 9: Example Snippet

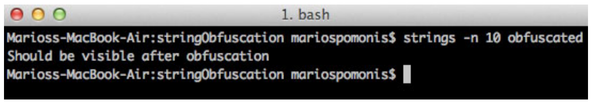
In this case b can be eliminated entirely from the source code since its only use is in the string comparison. As a result, someone that tries to disassemble the binary will not know it ever existed thus making his effort even harder.

In order to showcase this behavior we will use the program strings that returns all printable strings in a binary and ask it to return all printable strings of at least 10 characters.



```
Marioss-MacBook-Air:stringObfuscation mariospomonis$ strings -n 10 unobfuscated
Should NOT be visible after obfuscation
Should be visible after obfuscation
Marioss-MacBook-Air:stringObfuscation mariospomonis$
```

Figure 3: Printed message from unobfuscated (unmodified) executable



```
Marioss-MacBook-Air:stringObfuscation mariospomonis$ strings -n 10 obfuscated
Should be visible after obfuscation
Marioss-MacBook-Air:stringObfuscation mariospomonis$
```

Figure 4: Printed message from obfuscated version of executable

As we can see the string is visible in the unobfuscated file whereas after the pass it is removed.

8.2 Junk Code Insertion

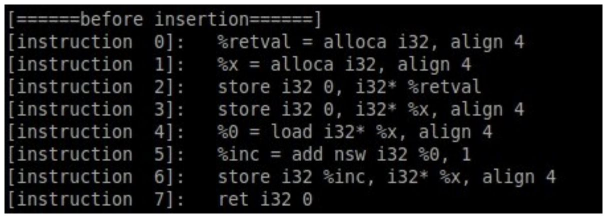
In general, we should add some instructions before a variable is stored, so we targeted on store instructions in junk code insertion. However, we do not have to insert junk code in every store instructions because it will not only slow the speed of programs but also easily be detected by reverse-engineer. To ease the problem, we focus only on integer, which is most widely used in arithmetic, for loop and while loop. The way to insert junk code are many. In this paper, we shows three different ways to insert the junk code.

1. $d * (x + e) = d * x + d * e$

2. $(ax + b) - (cx + b) = (a - c) * x$

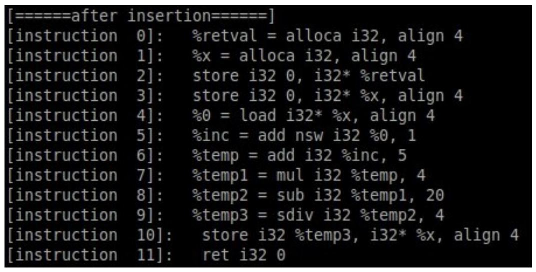
3. $(ax + b) - ((a - 1)x + c) = x + (b - c)$

The original three address code of snippet 3 is shown in Fig. 5. The result of each method is shown in Fig. 6(a), Fig. 6(b), and Fig. 6(c),respectively. Note that we do not deal with variable initialization (e.g., instruction 2 & 3 in Fig. 5) because its instruction is often at the beginning of the function, which makes obfuscation very distinct from the original code. For clarification, the junk code we inserted are those instructions begin with %**temp**. As we can see in Fig. 6(a) - 6(c), we added about four or five instructions.



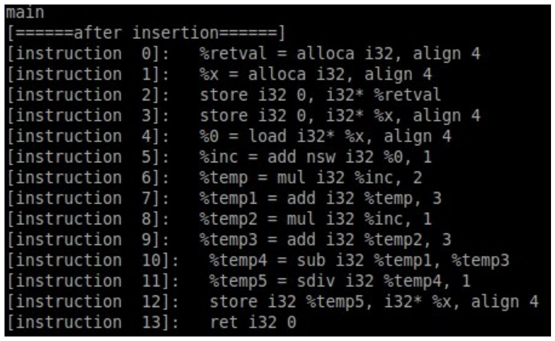
```
[======before insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   store i32 %inc, i32* %x, align 4
[instruction  7]:   ret i32 0
```

Figure 5: Original Three Address Code



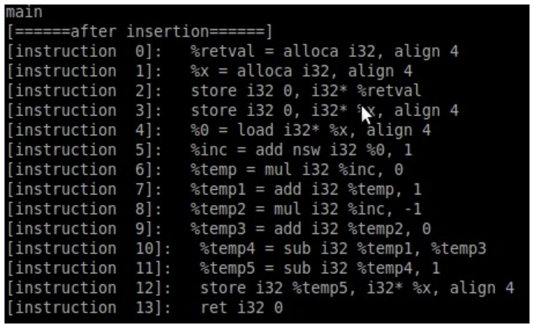
```
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   %temp = add i32 %inc, 5
[instruction  7]:   %temp1 = mul i32 %temp, 4
[instruction  8]:   %temp2 = sub i32 %temp1, 20
[instruction  9]:   %temp3 = sdiv i32 %temp2, 4
[instruction  10]:   store i32 %temp3, i32* %x, align 4
[instruction  11]:   ret i32 0
```

(a) Type1 insertion



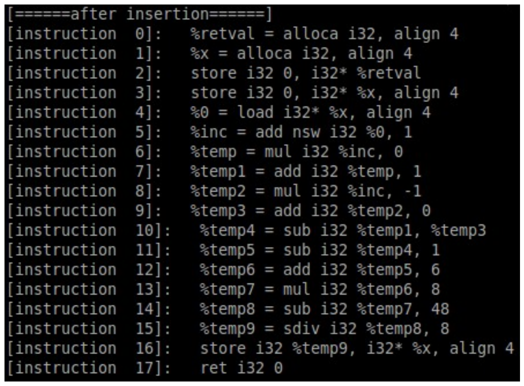
```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   %temp = mul i32 %inc, 2
[instruction  7]:   %temp1 = add i32 %temp, 3
[instruction  8]:   %temp2 = mul i32 %inc, 1
[instruction  9]:   %temp3 = add i32 %temp2, 3
[instruction  10]:   %temp4 = sub i32 %temp1, %temp3
[instruction  11]:   %temp5 = sdiv i32 %temp4, 1
[instruction  12]:   store i32 %temp5, i32* %x, align 4
[instruction  13]:   ret i32 0
```

(b) Type2 insertion



```
main
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   %temp = mul i32 %inc, 0
[instruction  7]:   %temp1 = add i32 %temp, 1
[instruction  8]:   %temp2 = mul i32 %inc, -1
[instruction  9]:   %temp3 = add i32 %temp2, 0
[instruction  10]:   %temp4 = sub i32 %temp1, %temp3
[instruction  11]:   %temp5 = sub i32 %temp4, 1
[instruction  12]:   store i32 %temp5, i32* %x, align 4
[instruction  13]:   ret i32 0
```

(c) Type3 insertion



```
[======after insertion======]
[instruction  0]:   %retval = alloca i32, align 4
[instruction  1]:   %x = alloca i32, align 4
[instruction  2]:   store i32 0, i32* %retval
[instruction  3]:   store i32 0, i32* %x, align 4
[instruction  4]:   %0 = load i32* %x, align 4
[instruction  5]:   %inc = add nsw i32 %0, 1
[instruction  6]:   %temp = mul i32 %inc, 0
[instruction  7]:   %temp1 = add i32 %temp, 1
[instruction  8]:   %temp2 = mul i32 %inc, -1
[instruction  9]:   %temp3 = add i32 %temp2, 0
[instruction  10]:   %temp4 = sub i32 %temp1, %temp3
[instruction  11]:   %temp5 = sub i32 %temp4, 1
[instruction  12]:   %temp6 = add i32 %temp5, 6
[instruction  13]:   %temp7 = mul i32 %temp6, 8
[instruction  14]:   %temp8 = sub i32 %temp7, 48
[instruction  15]:   %temp9 = sdiv i32 %temp8, 8
[instruction  16]:   store i32 %temp9, i32* %x, align 4
[instruction  17]:   ret i32 0
```

(d) Inserting junk code multiple times

In Fig. 6(d), we use Type2 insertion following by Type0 insertion, the junk code becomes more complicated than the original code.

8.3 Control Flow Obfuscation

We successfully managed to alter the control flow of the binary without altering its semantics, using the methods described section 5. We picked random spots to insert the opaque predicates inside the binary. The variables used as arguments in the equations forming the predicates are placed in such positions in the binary so that it won't not be easy for an adversary to infer that they are used in an opaque predicate with local static analysis. The insertion of the opaque predicate alters significantly the control flow graph, as it can be seen in Figures 6 & 7. These figures depict the connections between basic blocks for a simple if-statement, before and after obfuscation. It is clear that the flow graph after the obfuscation is more complex.

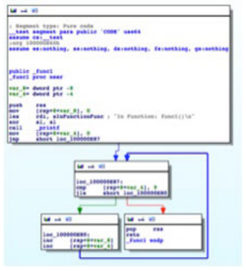


Figure 6: CFG outline before obfuscation

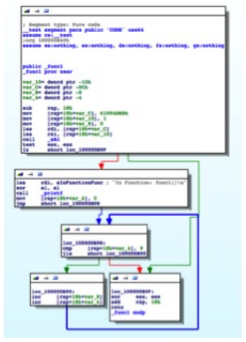


Figure 7: CFG outline after obfuscation

In order to evaluate the change in the complexity of the control flow graph as a result of our obfuscation, we define the **cyclomatic complexity (CC)** of a graph as follows[18]: For a graph G=(V,E) which consists of P connected components, the cyclomatic complexity M is

$$M = E - V + 2P$$

To measure the cyclomatic complexity of any given program, we wrote a Python script that runs on the IDA Pro Disassembler to compute the CC score from the disassembled program. Thus the CC is highly dependent on how well the IDA Pro Disassembler can disassemble a given program. This is a reasonable assumption since it approximates the effort of a reverse-engineer attempting to derive the semantics of a program using this disassembler.

In Figure 8 we present some results from the change in the cyclomatic complexity of test programs used in the evaluation phase of Confuse. We notice that the increase in the CC varies from 40% to 225%, depending on the operations in the binary.

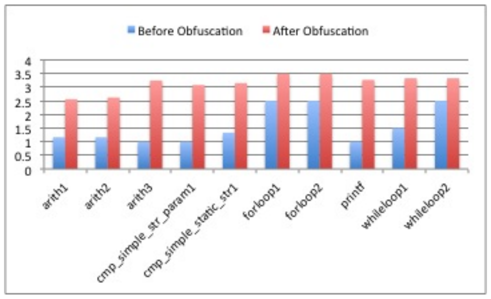


Figure 8: Cyclomatic Complexity Change

9 Conclusion

Confuse takes advantage of the LLVM IR and obfuscate the source code at compilation time. This is a very flexible scheme since it removes the unnecessary steps of compiling and then running an obfuscation application to the produced binary or assembly code. The implementation of the obfuscation in the form of optimization pass libraries allows users select and pick the obfuscation they want to use. It is also very scalable. More importantly, it allows our implementation to be universal for all the architectures and processor families supported by LLVM because we modify the LLVM IR and not the (architecture specific) binary code. To the best of our knowledge we are the first to create LLVM-based obfuscation tool for the aforementioned techniques. Balachandran et al [8] obfuscate at link time using C/C++ binary programs. Liem et al [7] use an front-end by Edison Design Group [19] and use Fabric++ as the intermediate representation. Finally the most similar work to ours is by Sharif et al [20] who have also used the LLVM IR, but employed different obfuscation techniques.

References

[1] Wikipedia. Reverse engineering, December 2012.

[2] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[3] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, 35(2):105–119, March 2009.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.

[5] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In

In DRM 03: Proceedings of the 3rd ACM workshop on Digital rights management, pages 142–153. ACM Press, 2003.

[6] Mila Dalla Preda and Roberto Giacobazzi. Control code obfuscation by abstract interpretation. In *In Proc. 32nd ICALP, LNCS 3580*, pages 301–310. IEEE Computer Society, 2005.

[7] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. A compiler-based infrastructure for software-protection. In *PLAS*, pages 33–44, 2008.

[8] V. Balachandran and S. Emmanuel. Software code obfuscation by hiding control flow information in stack. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1 –6, 29 2011-dec. 2 2011.

[9] Liang Shan and Sabu Emmanuel. Mobile agent protection with self-modifying code. *J. Signal Process. Syst.*, 65(1):105–116, October 2011.

[10] S.M. Darwish, S.K. Guirguis, and M.S. Zalat. Stealthy code obfuscation technique for software security. In *Computer Engineering and Systems (ICCES), 2010 International Conference on*, pages 93 – 99, 30 2010-dec. 2 2010.

[11] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007. USENIX Association.

[12] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[14] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, 9 2001.

[15] R. Rivest. The md5 message-digest algorithm. RFC 1321, 1992.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *IN PRINCIPLES OF PROGRAMMING LANGUAGES 1998, POPL98*, pages 184–196, 1998.

[17] Genevive Arboit. A method for watermarking java programs via opaque predicates. In *In Proc. Int. Conf. Electronic Commerce Research (ICECR-5*, 2002.

[18] Wikipedia. Cyclomatic complexity, December 2012.

[19] http://www.edg.com/.

[20] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. *Informatica*, 2008.

A Appendix

A.1 Usage

We have designed each obfuscation technique to be in the form of an optimization pass within one single library *Obfuscation.so*. Each obfuscation technique can be invoked as a pass by passing in its corresponding flag *-obstring*, *-objunk* or *-opaque_predicates*. To make a turn-key solution to transform and compile a given *.c* source file to an obfuscated binary executable, we wrote a Makefile that allows us to invoke specific obfuscation, or apply all the obfuscation passes together.

In the *confuser* directory, use the following command to perform individual obfuscation on a given *.c* source file, with filename *test.c*. This will create two sets of binary and readable LLVM IR listing, one each for the original code and obfuscated code. The files can be found in the directories *original* and *confused* respectively. In this way, we can easily compare the differences between the original code and the generated obfuscated version.

```
make [obstring|obop|objunk] TARGET=test
```

To use all the obfuscation passes on a given file, use the following command:

```
make all TARGET=test
```

In Section 8.3, we describe using the CC score as a means to measure the approximate complexity of the control flow graph of a program. We augmented the Makefile to perform the disassembly and CC computation of the programs automatically by invoking the IDA Pro Disassembler and the Python script. This may take a while if the program is large. We can perform this CC evaluation by using the following command. The CC score is computed for each function of a given program. The average, minimum, and maximum CC scores of the functions in the program will be displayed for both the original and obfuscated versions.

```
make evalcc TARGET=test
```

A.2 Lessons Learnt

A.2.1 Chih-Fan Chen

I think the most important thing that I learned is not the context of obfuscation but how to work with other people. Before the project, I know nothing about obfuscation. In the beginning, I was afraid that I cannot catch up with the team. I am glad that I have very good mentor and nice teammates to help me through. Though this project, I learned how to use the LLVM to read, create, and change the instructions. I know more about how the complier works and how we can use some technique to obfuscate others from reassemble the code. By comparing the project and the materials learned from class, I think it is better than learning from instructors speech.

A.2.2 Theofilos Petsios

Through this project I had the chance to have a greater understanding of the inner parts of a compiler, the optimization phases and the considerations that have to be taken into account for maintaining the semantics any source code that gets compiled. It was made clear to me how important regression testing is in software engineering. I also learned a lot about LLVM and how to deliver a project in steps, being part of a small team, and also about how to present your results and ask questions on the pros and cons of your software. Overall, the guidance from the teaching staff and my teammates was a great aid. This project was a lot of fun and I believe we all have build a good foundation to continue this work in a research level.

A.2.3 Marios Pomonis

This project gave me the opportunity to realize the importance of the unseen hero of the compilation process: the IR. Before we started I considered having a strong back-end the most crucial part of the compiler however this changed drastically because I realized that without a well designed IR that will allow effective transformation passes, the back-end will only be able to make minor optimization when producing assembly or the machine code.

I was also given the chance to study the internals of the LLVM framework since I had never bothered doing it in the past. A future task that I have set for myself is to do the same for the GCC something that illustrates my amazement about those that I learned in this process.

Last but not least, I was able to verify once again that given a hard-working and talented core of teammates any task becomes exponentially easier and simpler, especially if team chemistry is there. Guidance and support from the teaching staff is also vital. Since this was my first team project in Columbia, I believe it is worth sharing.

A.2.4 Adrian Tang

Through this project, I have observed that the IR level seems to be an appropriate granularity to obfuscate a program given the availability of the source code. Working at this phase has a good balance of being high-level enough to have enough language semantic encoded in the IR to perform meaningful obfuscation and yet low-level enough to ensure the obfuscation persists to the target machine code. I have learnt a tremendous amount ranging from the technical bits like the inner workings of LLVM and dissecting a program with disassembler, to the general aspects of managing and collaborating in a project team. The learning curve was steep but a great deal of fun. I also appreciate all the support and advice the teaching staff has given us. I believe this project has given us some foundation in the workings of the compiler that will certainly help us in the further pursuit of research in this area.

A.3 Future Works

Code obfuscation is still an active research topic of both academic and industry interest. While creating **C**onfuse we were given the opportunity to familiarize ourselves with many techniques proposed in the bibliography. To the best of our knowledge only a small portion of the researchers implements their obfuscation in the optimizer level which in our opinion yields significant benefits (with portability and flexibility being the keywords in this case), thus we believe that if **C**onfuse were to be continued (as a research project) and be enhanced with a novel technique that would take advantage of the its design it might produce a research paper.

Given the scope of the project, we regret not being able to conduct a more comprehensive evaluation of the overhead introduced to an obfuscated binary. This is certainly an area that we will look into for any future work.