



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

## ASSIGNMENT OF MASTER'S THESIS

**Title:** LLVM Obfuscator  
**Student:** Bc. Martin Petráček  
**Supervisor:** Ing. Tomáš Zahradnický, Ph.D.  
**Study Programme:** Informatics  
**Study Branch:** System Programming  
**Department:** Department of Theoretical Computer Science  
**Validity:** Until the end of summer semester 2018/19

### Instructions

Explore the taxonomy of obfuscation transformations [1]. Get acquainted with the LLVM framework [2] and its internal representation (IR) format. Use generated IR to create a Control Flow Graph (CFG) of each function. Design and implement a set of obfuscation transformations taking IR or CFG on the input and producing an obfuscated IR on the output. A minimum set of transformations shall include outlining, inlining, and table representation. Analyze potency and resilience of each implemented transformation. Compare the results with other available obfuscation tools.

### References

1. Collberg C. et al. A Taxonomy of Obfuscation Transformations. Technical Report #148. University of Auckland. New Zealand. <https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf>
2. LLVM Team. The LLVM Compiler Infrastructure. University of Illinois at Urbana-Champaign. <http://llvm.org/>.

doc. Ing. Jan Janoušek, Ph.D.  
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.  
Dean

Prague February 6, 2018





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Master's thesis

## **LLVM Obfuscator**

*Bc. Martin Petráček*

Department of Theoretical Computer Science  
Supervisor: Ing. Tomáš Zahradnický, EUR ING, Ph.D.

May 8, 2018



---

## **Acknowledgements**

I would like to thank my supervisor, Ing. Tomáš Zahradnický, EUR ING, Ph.D. for his enthusiasm, patience and helpful advices.

I would like to also thank my family and friends who supported me throughout my studies and encouraged me when times were tough.



---

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 8, 2018

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2018 Martin Petráček. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Petráček, Martin. *LLVM Obfuscator*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.



---

# Abstrakt

Obfuskace je známá technika pro ochranu duševního vlastnictví obsaženého v software. Obfuskace softwaru může být prováděna ručně vývojáři, ale to je časově náročné a omezuje to jeho udržitelnost. Domníváme se, že lepším přístupem je provádět obfuskaci automaticky, jako součást procesu kompilace. Modularita populárního kompilátoru LLVM nám dává možnost toto udělat. Tato práce je zaměřena na implementaci několika obfuskáčnických transformací do LLVM a popisuje výhody a omezení tohoto řešení.

**Klíčová slova** obfuskace, transformace, LLVM, potency, resilience, performance

---

# Abstract

Obfuscation is a method for protecting intellectual property contained within software. Obfuscation can be performed manually by developers, but that is time consuming and it limits maintainability of the software. We assume that it is better to perform obfuscations automatically, as a part of compilation process. The modularity of popular compiler LLVM makes it possible to implement that. This work is focused on implementing several of these transformations and describes the advantages and limitations of this approach.

**Keywords** obfuscation, transformations, LLVM, potency, resilience, performance

---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Obfuscation transformations</b>	<b>3</b>
1.1 Basic concepts . . . . .	4
1.2 Evaluating obfuscations . . . . .	6
1.3 Obfuscations overview . . . . .	8
<b>2 State-of-the-art</b>	<b>17</b>
2.1 CXX-OBSUF . . . . .	17
2.2 StarForce C++ Obfuscator . . . . .	17
2.3 Tigress C Obfuscator . . . . .	18
2.4 Obfuscator-LLVM . . . . .	18
2.5 Summary . . . . .	19
<b>3 Design</b>	<b>21</b>
3.1 LLVM Framework . . . . .	21
3.2 Implementation design . . . . .	25
<b>4 Implementation</b>	<b>35</b>
4.1 Implementation details . . . . .	35
4.2 Usage and limitations . . . . .	38
<b>5 Evaluation</b>	<b>41</b>
5.1 Obfuscation metrics . . . . .	41
5.2 Comparison with other obfuscators . . . . .	54
<b>Conclusion</b>	<b>57</b>
Possible future work . . . . .	57
<b>A Acronyms</b>	<b>59</b>

<b>B Configuration options</b>	<b>61</b>
B.1 Scheduler pass . . . . .	61
B.2 Obfuscation passes . . . . .	61
<b>Bibliography</b>	<b>63</b>
<b>C Contents of enclosed medium</b>	<b>65</b>

---

## List of Figures

1.1	Control flow graph example . . . . .	4
1.2	Code insertion CFG examples . . . . .	9
1.3	Table interpretation CFG example . . . . .	11
1.4	Inlining code example . . . . .	12
1.5	Outlining code example . . . . .	13
1.6	Outlining region example . . . . .	14
1.7	Function interleaving code example . . . . .	15
3.1	3-phase compiler architecture . . . . .	21
3.2	LLVM IR example . . . . .	22
3.3	Splitting basic block example . . . . .	27
3.4	Interleaved function CFG example . . . . .	30
3.5	Table interpretation - LLVM IR example . . . . .	32
5.1	Visualization of table interpretation effect in IDA . . . . .	45
5.2	Slowdown caused by splitting basic blocks . . . . .	49
5.3	Slowdown caused by obfuscating inner loops . . . . .	49
5.4	Matrix multiplication performance . . . . .	51
5.5	Matrix multiplication performance - relative . . . . .	51
5.6	Merge sort performance . . . . .	52
5.7	Merge sort performance - relative . . . . .	52
5.8	AES performance . . . . .	53
5.9	AES performance - relative . . . . .	53



---

## List of Tables

1.1	Invariant opaque predicate examples . . . . .	5
1.2	Contextual opaque predicate examples . . . . .	6
5.1	Suggested obfuscation properties . . . . .	41
5.2	Program size changes . . . . .	42
5.3	Computational complexity changes . . . . .	43
5.4	Evaluated obfuscation properties . . . . .	54





---

# Introduction

For software vendors, it is essential to protect their intellectual property. They may want to ensure that their product will be used only by legitimate users and will not be subject to software piracy or copying. Or they may want to protect some assets of their product, such as an innovative algorithm. In that case, they might try to modify their software to be harder to disassemble and reverse engineer. Various solutions to this problem exist, with different advantages and disadvantages.

One solution that is gaining popularity nowadays is to keep the main part of the valuable software solution on the server side and let the user use it remotely, over the Internet. Users only need a thin client application or no specific application at all (using an Internet browser, that is already bundled with their system). This approach is referred to as Cloud services or as a Software as a Service (SaaS). That significantly simplifies the need to protect assets within the software, as the user does not have access to actual software running on the server side.

While this may be convenient for some applications, it does not suit all use cases. It requires users to be always online in order to use the service. The network bandwidth might be an issue. Also, users have to fully trust the service provider, as all the data is transferred to and processed on their servers. In some cases, that could be a problem, so it is still common to use software running on local machine. In that case, software developers have to think how to protect programs they distribute to users.

Protection methods can generally be divided into hardware and software protection. Hardware methods require the use of specialized hardware to run the program. It might be a special trusted processor to prevent tampering or a dongle (nowadays a special USB stick), that needs to be plugged to allow running software. In many cases, however, the user cannot be convinced to use a specialized hardware in order to use the program, so the developers need to revert to software protection methods.

Software methods do not require users to have any specialized hardware.

They modify the program before shipping it to a user, with the aim to make it more resistant to reverse engineering and tampering attempts. Software protection methods are usually divided into three categories: obfuscation, tampering resistance and watermarking. Obfuscation changes the program, so it is harder to understand and analyze by means of reverse engineering. Tampering resistance is used to detect user's attempts to modify the program. Finally, watermarking can be used to create slightly different versions for different customers - so when the program leaks, it is possible to identify the source of the leak.

This work focuses on program obfuscation. It is important to note that it is impossible to achieve perfect protection by obfuscation. Given enough time and determination, a skilled reverse engineer can always analyze and understand the program. Their task can, however, be made much more difficult. When the difficulty exceeds a certain threshold, the belief is that they may give up, as reverse engineering the program may not be worth the effort.

A survey of obfuscation transformation was given in the work of Collberg et al.[1]. Obfuscations can be performed manually on the source code by developers, but that is time-consuming, and it limits the maintainability of the software (as we will see later, obfuscation is quite the opposite to making program maintainable). We believe that a better approach is to obfuscate the program automatically, as a part of the compiler toolchain.

The aim of this work is to implement an automatic obfuscator. To do that, we will use LLVM compiler infrastructure. LLVM compiler is one of the most popular nowadays and its modularity allows to do custom transformation as a part of the compilation process. The advantage of LLVM is also that it supports many programming languages (C/C++, Go, Rust, Fortran) and multiple CPU architectures. Its architecture allows the obfuscation transformations which we will create to be completely independent on source language or target architecture.

The rest of this work is structured as follows: Chapter 1 introduces and categorizes obfuscation transformations. Chapter 2 presents some existing obfuscators. Chapter 3 describes design of our obfuscator. Chapter 4 describes its implementation and implementation issues. Chapter 5 evaluates implemented transformations and compares our obfuscator to existing obfuscation tools.

---

# Obfuscation transformations

In this section, we give an overview of obfuscation transformations and metrics to evaluate them. First, we define what an obfuscation transformation is. Informally, we can say that obfuscation should make the program more difficult to reverse engineer while preserving functionality. The notion of obfuscation transformations was formalized by Collberg et al. in [1]:

**Definition 1** *Let  $P \xrightarrow{\tau} P'$  be a transformation of a source program  $P$  into a target program  $P'$ .  $P \xrightarrow{\tau} P'$  is an obfuscating transformation, if  $P$  and  $P'$  have the same observable behavior. More precisely, in order for  $P \xrightarrow{\tau} P'$  to be a legal obfuscating transformation the following conditions must hold:*

- *If  $P$  fails to terminate or terminates with an error condition, then  $P'$  may or may not terminate.*
- *Otherwise,  $P'$  must terminate and produce the same output as  $P$ .*

Collberg et al. in [1] categorized obfuscations by the kind of information they target. They divided these transformations into 4 basic categories:

1. **Layout obfuscations** remove useful information from the source code, e.g., removing debugging information or scrambling identifier names.
2. **Preventive obfuscations** make automatic deobfuscation more difficult, e.g., by adding bogus data dependency to obfuscated constructs to prevent simplifying them or targeting some known weakness of a particular deobfuscator.
3. **Data obfuscations** target data and data structures used in the program, e.g., promoting variables to more general type, splitting variables, or restructuring arrays.
4. **Control flow obfuscations** break the flow within the program, e.g., insert bogus conditional branches and unreachable or redundant code, inlining to and outlining from functions.

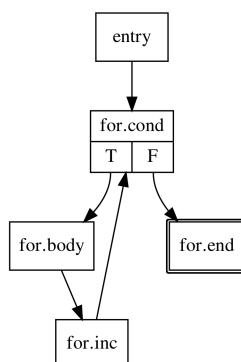


Figure 1.1: Control flow graph example

In our work, we focus on control flow obfuscation. A detailed description of these obfuscations is given later in this chapter. First, we need to explain some basic concepts important for control flow obfuscations. Then, we describe some metrics for evaluating obfuscations. Finally, we provide a detailed description of several control flow obfuscations.

## 1.1 Basic concepts

### 1.1.1 Control flow graph

A control flow graph (CFG) is a directed graph representing a function. CFGs were studied extensively as a part of compiler design[2]. A CFG represents all possible paths within a function. Each graph node (a basic block) contains a linear sequence of instructions. Only the last instruction of a basic block (BB) can jump to another BB or return from a function. Jumps are allowed only to the first instruction of a BB. Thus, once the first instruction in a BB starts execution, it's guaranteed that all instructions in its BB will be executed (unless an exception such as division by zero occurs).

The last instruction (called terminator instruction) defines direct successors of the BB. There can be one successor (unconditional jump), more successors (destination depends on some condition) or no successor (BB returns from a function) at all. All basic blocks that jump to given basic block are predecessors of that basic block. Each function entry block may not have any predecessor while each function terminating block never has a successor.

One important concept related to a CFG is dominance. A basic block  $A$  is said to dominate  $B$  if any path from the *entry* basic block to  $B$  must go through  $A$ .

Figure 1.1 gives a simple example of a CFG of a function. Instructions inside each BB are not important here, so they were omitted. We can, for example, see that (the only) successor of *entry* is *for.cond*. Predecessors of

Table 1.1: Invariant opaque predicate examples[6]. All these expression are always true.

Expression
$\forall x \in \mathbb{Z}, x^2 \geq 0$
$\forall x \in \mathbb{Z}, 2 x(x+1)$
$\forall x \in \mathbb{Z}, 2 \lfloor \frac{x^2}{2} \rfloor$
$\forall x \in \mathbb{Z}, (x^2 + 1) \% 7 \neq 0$
$\forall x \in \mathbb{Z}, (x^2 + x + 7) \% 81 \neq 0$
$\forall x \in \mathbb{Z}, (4x^2 + 4) \% 19 \neq 0$
$\forall x, y \in \mathbb{Z}, 7y^2 - 1 \neq x^2$

*for.cond* are *entry* and *for.inc*. Also, we can see that *for.body* dominates *for.inc*, but does not dominate any other basic block.

### 1.1.2 Opaque predicates and variables

Opaque predicate is an important concept for designing obfuscations. Let us have a variable such that its value is known at the obfuscation time, but it is difficult to deduce for deobfuscator. This is known as an opaque variable[1]. Opaque predicate is a just a boolean opaque variable. Creating opaque variables and predicates that are difficult to guess is a major challenge of obfuscator design and it is the key to creating difficult-to-remove obfuscations.

A lot of research was dedicated to designing opaque predicates [3, 4, 5] and similar research interest was dedicated to identifying them[6, 7]. The current state-of-art tool for detecting opaque predicate is LOOP[6]. The article identified three types of opaque predicates.

The first type is an invariant opaque predicate. An invariant opaque predicate always evaluates to true or false, for all possible inputs, but only the obfuscator knows this value at compile time. Many of them are derived from known algebraic theorems or quadratic residues. Table 1.1 shows several examples of these. However, the invariant property is also the drawback of this type of opaque predicates – a constraint solvers can identify that their value is always the same[6, 7].

Article [8] proposes another opaque predicate type. It uses a formula, that is only true under a specific precondition. But it can be false if this precondition does not hold. Table 1.2 gives some examples of these predicates. The precondition might be hidden in another parts of code or implicitly known to the obfuscator. In article [6], they call this type contextual opaque predicate. Constraint solvers cannot detect it by analysing individual expressions, they have to consider also the context of those expressions.

The third and last type described in [6] is called dynamic opaque predicate. It consists of a set of correlated predicates, meaning that all present the same

Table 1.2: Contextual opaque predicate examples[6]. Expressions are true, if the precondition holds. They can be false otherwise.

Precond.	Expression
$\forall x \in \mathbb{Z}, x > 5$	$x \geq 0$
$\forall x \in \mathbb{Z}, x > 3$	$x^2 - 4x + 3 > 0$
$\forall x \in \mathbb{Z}, x \% 4 = 0$	$x \% 2 = 0$
$\forall x \in \mathbb{Z}, 3 (7x - 5)$	$9 (28x^2 - 13x - 5)$

value, but this value may vary in different runs – the value of such predicates switches dynamically. However, it should be noted that this type of predicate is not a general-purpose predicate as the previous types. To our knowledge, this type can be only used for inserting redundant code, as described in 1.3.1.1.

In the following sections, we will use  $P^T$  to denote opaque predicate that always evaluates to true and  $P^F$  for a predicate that always evaluates to false. Note that we can easily make  $P^T$  from  $P^F$  just by negating it and vice versa.

## 1.2 Evaluating obfuscations

To be able to assess the quality of different obfuscations, we need some metrics. Several those metrics were proposed by Collberg et al. in [1]. In their work, they evaluated obfuscation transformation based on 3 criteria: (i) how much obscurity they add to the program (potency), (ii) how hard they are to remove/break (resilience) and (iii) how much computational overhead they add to the program.

### 1.2.1 Potency

The measure of potency describes how much more difficult (for a human being) the program is to understand than a non obfuscated program. Informally, we can say that that the obfuscation is potent if it does a good job confusing reverse engineers by hiding the intent of the original code. This is not easy to measure precisely since that depends on their cognitive abilities.

Collberg et al. in [1] suggested using software complexity metrics from software engineering field. These metrics were designed to aid keeping the program readable and maintainable. We have chosen two of those metrics:

- **program length** ( $\mu_1$ ): a number of operators and operands in a function/program
- **cyclomatic complexity** ( $\mu_2$ ): a number of independent paths though a function/program

Software engineers tries to keep these metrics low, in order to keep the code readable and mainainable. However, the aim of obfuscation is quite the

opposite – it aims to make the code hard to read. Thus, they are trying to increase these metrics. Based on that, we can list some desirable properties of obfuscation transformation. For example, a potent transformation can:

- increase the size of the program (thus increasing  $\mu_1$ )
- increase the number of paths through a function (increasing  $\mu_2$ )

Note that the increase of those metrics by obfuscation should be reasonable. It would be possible to increase program size unlimitedly, but the user likely expects a program of a reasonable size.

### 1.2.2 Resilience

From the previous section, it may seem that it is easy to obfuscate a program – for example just by adding a lot of dead code (a code that does not contribute to any result) to each function. However, such a code can be easily detected automatically. A well known optimization - dead code elimination - aims to detect and remove such code.

Potency does not measure how difficult is to break the obfuscation for an automatic deobfuscator. For that, another metrics is needed – it is called resilience. Resilience in [1] is seen as a combination of two factors: i) the amount of time needed for a programmer to construct an automatic deobfuscator and ii) execution time and space needed by the deobfuscator to remove the obfuscation. They measure resilience on a five-point scale from trivial to weak, strong, full, and one-way. One-way obfuscation transformations are special, as they cannot be undone. Such transformations remove some information from the program. Other transformations can be removed with varying level of difficulty.

Programmer’s effort, the work to create a deobfuscator for a transformation is seen as a function of the scope of that transformation. The scope could be:

1. **local**: if the transformation affects only a single BB
2. **global**: if it affects an entire CFG of a function
3. **inter-procedural**: if it affects the flow of information between functions
4. **inter-process**: if it affects the interaction between execution threads

Resilience of a transformation corresponds to the scope of such transformation – local transformation having trivial resilience, global transformation having weak resilience, inter-procedural having strong resilience and inter-process having full resilience<sup>1</sup>.

---

<sup>1</sup>if the created deobfuscator requires polynomial time and space

### 1.2.3 Performance impact

The obfuscated program often needs more resources than the not obfuscated one. It is usually bigger and takes more time to execute. It is, therefore, necessary to consider this performance penalty as well. The selection of obfuscation transformations is usually a trade-off between performance penalty caused by the obfuscation and the protection gained by the obfuscation. In [1], they measured performance cost on a four-point scale:

- **free**: if executing obfuscated program required  $O(1)$  more resources than non obfuscated program
- **cheap**: if executing obfuscated program required  $O(n)$  more resources than non obfuscated program
- **costly**: if executing obfuscated program required  $O(n^p), p > 1$  more resources than non obfuscated program
- **dear**: if executing obfuscated program required exponentially more resources than non obfuscated program

## 1.3 Obfuscations overview

In [1], control flow obfuscations were divided into three categories:

1. **Computation obfuscations**: Insert new (dead or redundant) code into the program or make changes of the algorithm.
2. **Aggregation obfuscations**: Break up code pieces that belong together or merge pieces that do not.
3. **Ordering obfuscations**: Changes order in which the operations are performed.

We will explore these obfuscations in bigger detail in the following sections.

### 1.3.1 Computation obfuscations

Computation transformations insert new code (dead or redundant) into the program. While adding new instructions, care has to be taken to not interfere with any existing instructions. A trivial dead code can be easily identified - operations that write variables that do not contribute to the result are normally removed by the compiler. To make these transformations more resilient, inserted dead code must be hard to identify and remove. This is where opaque predicates can help.



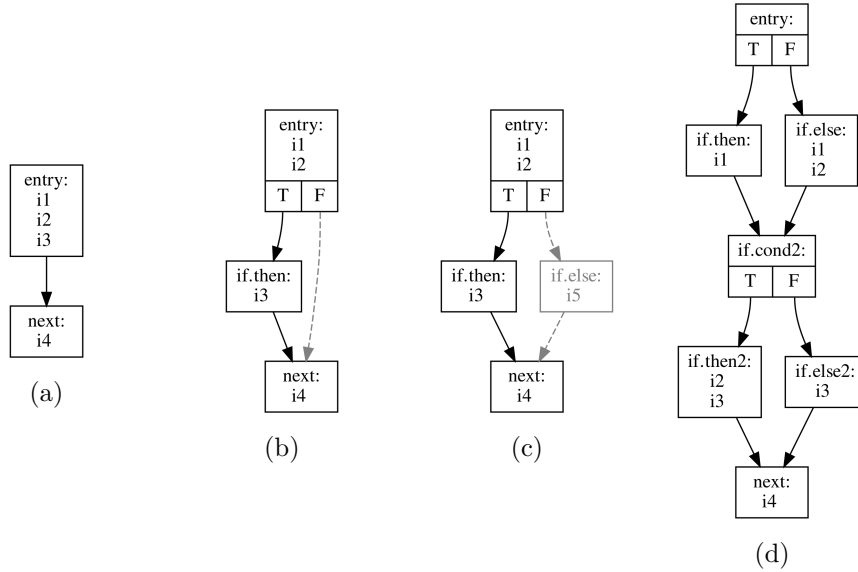


Figure 1.2: Code insertion examples. (a) is the original CFG. (b) shows CFG where bogus jump were added. (c) shows CFG where bogus jump and dead block was added. (d) illustrates usage of dynamic opaque predicates. Grayed edges and blocks are never taken.

### 1.3.1.1 Inserting dead or redundant code

Metric  $\mu_2$  defined in 1.2.1 suggests that there is a strong correlation in between the perceived code complexity and the number of possible code paths. This obfuscation exploits that by inserting additional branches into the CFG. The executed instructions order is not altered. Newly introduced basic blocks are either dead or redundant. Opaque predicates can be used to ensure that. This obfuscation is also known as Bogus Control Flow.

There are many possible ways how code can be inserted. Figure 1.2 shows several simple examples on a straight code line (within a basic block). Assume that 1.2(a) shows the original state, before the obfuscation. Basic block *entry* contains three instructions and a jump to the *next* basic block. This transformation has to guarantee that the correct sequence of instructions will always be executed.

1.2(b) changes the flow by adding one opaque predicate  $P^T$ . This guarantees that the flow continues always in the correct direction (the false branch is never taken). The opaque predicate should ensure that this fact is not obvious to the deobfuscator.

1.2(c) shows another possibility with dummy instruction added to the unreachable branch. Branches can be also swapped by using  $P^F$ . If the analysis cannot detect that the predicate is opaque, it needs to consider both control paths.

Yet another possibility is by using dynamic opaque predicates. Dynamic opaque predicates are two correlated predicates – they both contain the same value, but the value may vary in different runs. 1.2(d) shows an example. In any run, both conditions (in *entry* and *if.cond2*) evaluate to the same value. This ensures that the same sequence of instructions is always executed.

Potency and resilience of this transformation depends on the used opaque predicate. If a potential reverse engineer quickly finds out that the branch predicate has always the same value, they will quickly see through this obfuscation and ignore dead branches. Otherwise, the number of code paths will grow quickly. It is important how well the opaque predicate hides in the other code. Resilience is entirely dependent on the opaque predicate. The current state-of-the-art opaque predicate detector [6] can detect many predicates we described in 1.1.2. However, in [4], Xu et al. states that creating more obscure control flow structures by using dynamic opaque predicates can counter its detection.

### 1.3.1.2 Remove library calls

Programs largely depend on the usage of standard library functions. These library functions can provide useful clues to a reverse engineer. For example, library functions to open files may help a reverse engineer to quickly find code parts where some files are read. It may be then traced where the read values are stored, how the execution depends on them and so on. These clues can be removed by simply providing own version of the standard library.

In [1], this obfuscation was said to have medium potency and strong resilience. Both potency and resilience might be increased by further use of inlining, that would break the abstraction presented by those functions. Inlining of dynamically linked libraries is otherwise not possible (library functions are external symbols, their bodies are not defined in the program).

### 1.3.1.3 Table interpretation

Table interpretation completely hides the real control flow. BBs are assigned a number and one basic block (called a *switch* or *dispatcher*) is added. This BB is provided a sequence of numbers and ensures that BBs are executed in the correct order. The real control flow is not apparent anymore – every BB is a successor of the *switch* and jumps to the *switch* again. Collberg et al. stated in [1] that table interpretation is one of the most effective (but expensive) transformations. In different sources, this obfuscation is also called dynamic dispatcher [9] or control flow flattening [10, 11].

Figure 1.3 shows a very simple example. The *switch* basic block has been added and the original control flow has been broken up. At the end of each BB (except the *switch*), an index of the successor basic block is stored, e.g. *init* stores the index of *for.cond*, *for.cond* stores either the index of *for.cond*

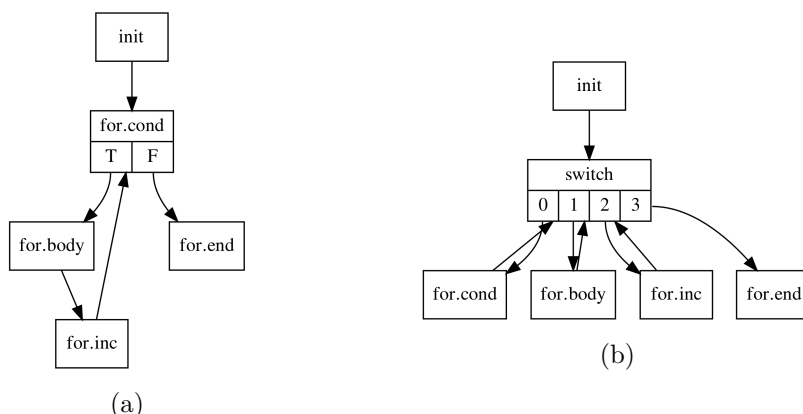


Figure 1.3: Example of table interpretation. (a) shows the original control flow. (b) shows the result.

or *for.end* (depending on the predicate value used previously for the conditional branch). In *switch*, this index is loaded and the next block is chosen accordingly.

This transformation will introduce some performance penalty because of the additional branches and repeated execution of the *switch* block. Every former jump to a successor of a basic block is replaced by a jump to the dispatcher block, that is then followed by a jump to the actual successor.

The potency and resilience of this transformation depend on the method of choosing the next block. If that would be done just by simply assigning integers as in the figure, that would be easy to break. This can be furthermore complicated by using opaque predicates or variables to choose the next location. Chow et al. in [10] suggested splitting the basic blocks into smaller pieces and adding dummy states to make the analysis harder.

### 1.3.2 Aggregation obfuscations

The motivation behind aggregations is to break up abstractions created by the programmer. We can find abstractions on many levels in the program – one of them is the procedural abstraction. Developers tend to create abstractions by grouping code that logically belongs together into functions. With that assumption in mind, aggregation obfuscations should:

- break code aggregated in a function and scatter it over the code;
- aggregate code that does not seem to belong together into one method.

#### 1.3.2.1 Function inlining

Function inlining is a well-known compiler optimization. Compilers try to automatically inline some functions to improve performance.

```
int min(int a, int b){
    if (a<b) return a;
    else return b;
}
int max(int a, int b){
    if (a>b) return a;
    else return b;
}
int main(){
    ...
    a=min(n, 100);
    b=max(n, 0);
    ...
}
```

(a)

```
int main(){
    ...
    if (n<100) a=n;
    else a=100;
    if (n>0) b=n;
    else b=0;
    ...
}
```

(b)

Figure 1.4: Inlining example. (a) shows the original code, (b) shows the code after inlining. Calls to *min* and *max* are being inlined into *main* and the original *min* and *max* functions removed. Note that this example is trivial and the compiler would probably do it itself, during the compilation.

Inlining as an obfuscation is very useful because it removes the abstraction introduced by developers. It is very resilient – it is essentially one-way – when a call is replaced by a function body and the function itself is removed, there are no traces of the abstraction left in the program. Inlining is a cheap transformation, it does not add any useless operations. There might be some indirect performance penalty, caused by worse cache behavior, but that is not too significant. Inlining may, on the other hand cause the code to grow significantly and therefore it must be done reasonably.

In [1], they described function inlining as having medium potency and one-way resilience. If the function is deleted after inlining, it cannot be recovered.

### 1.3.2.2 Function outlining

Outlining is another obfuscation that breaks the abstractions created by developers. Outlining extracts pieces of code grouped in one function into several functions – it aims to scatter related pieces of code to several places. It is a contrary transformation to inlining – once some functions are inlined into their call sites (and their procedural abstraction is removed), it is useful to extract some previously unrelated pieces of code into separate function to create a false procedural abstraction.

While it is essentially possible to inline without any limitation, it is way harder with outlining. For outlining, a region of code has to be selected. A region is a piece of control flow graph that has the following properties:

---

```

int main(){
    ...
    if (n<100) a=n;
    else a=100;
    if (n>0) b=n;
    else b=0;
    ...
}

```

(a)

```

void outlined(int a1, int a2, int a3,
int * a4, int * a5){
    if (a1<a2) *a4=a1;
    else *a4=a2;
    if (a1>a3) *a5=a1;
    else *a5=a3;
}
int main(){
    ...
    outlined(n, 100, 0, &a, &b);
    ...
}

```

(b)

Figure 1.5: Outlining example. (a) shows the original code, (b) the code after outlining. Some code from *main* is extracted into new function. Note that values used in the outlined function needs to be passed as arguments (*a1*, *a2* and *a3*). Also, values defined in the outlined function that are used in the original function need to be returned (*a* and *b* from *main*) – they are passed as pointers to outlined function (*a4* and *a5*).

1. Has a single block dominating all the others in the region (*entry* block).
2. No other block then *entry* has a predecessor that is not a part of the selected region.

These properties ensure that there is a single block, where the block outside of the region may jump. Other blocks in the region may then be extracted safely. This also ensures that the region is continuous, i.e. that there are no blocks in the middle of the region that are not a part of the region. Figure 1.6 tries to illustrate these conditions.

In [1] this obfuscation was evaluated as having medium potency and strong resilience

### 1.3.2.3 Function interleaving

Function interleaving merges several functions into one. In essence, this obfuscation takes several different functions, merges their bodies and arguments and adds an argument to distinguish between those functions. Merged function then has the functionality of several functions and the actual functionality is chosen by one argument. Calls to original functions are then replaced and original functions are removed.

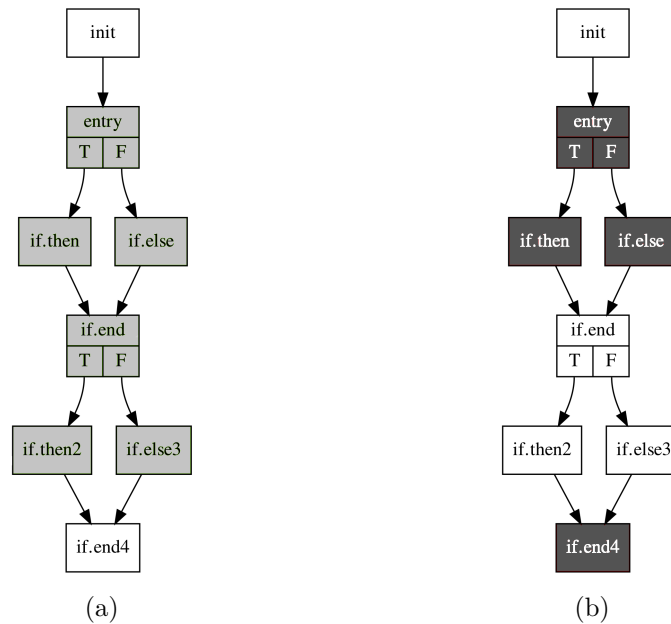


Figure 1.6: Outlining region example. Selected blocks are filled. In (a), the selected blocks are eligible for outlining, since it meets both conditions. In (b) the selected blocks cannot be outlined into a new function, as they violate the second condition (e.g., *if.end4* has a predecessor *if.then2* that is not part of the selected region).

It is advantageous if the merged functions have similar parameters, that allow reusing them for different functionalities. The code may resemble code handling some special cases of parameters.

Potency and resilience of interleaving depend on whether a potential reverse engineer would recognize that function has been created by interleaving several functions. In [1], they suggest to use opaque predicates to protect the argument to select functionalities and say that both potency and resilience depend on the used opaque predicate.

#### 1.3.2.4 Function cloning

If a function is called from multiple places in a program, we can confuse the reverse engineer by cloning the function and replacing some of the calls by calls to the cloned function. Then one may think that different functions (with a different behavior) are called, when, in fact, it is actually not the case.

Potency depends on how hard it is for a potential reverse engineer to recognize that the functions are identical. If the functions would be just cloned without any changes, it would likely be easy. The potency can be increased by modifying one of them in a different way or for example by changing the order of arguments.

---

```

int min(int a, int b){
    if (a<b) return a;
    else return b;
}
int max(int a, int b){
    if (a>b) return a;
    else return b;
}
int main(){
    ...
    a=min(n, 100);
    b=max(n, 0);
    ...
}

```

(a)

```

int interleaved(int a1, int a2, bool a3){
    if (a3) {
        if (a1<a2) return a1;
        else return a2;
    } else {
        if (a1>a2) return a1;
        else return a2;
    }
}
int main(){
    ...
    a=interleaved(n, 100, 1);
    b=interleaved(n, 0, 0);
    ...
}

```

(b)

Figure 1.7: Function interleaving example. (a) shows the original code, (b) shows the code after interleaving. Two functions *min* and *max* are merged together, creating a single function *interleaved*. Since the functions have the same arguments, the *interleaved* function have the same number of arguments. An additional argument is added to choose the requested functionality.

The same holds for resilience. If the functions would be identical, it would be easy for a deobfuscator to detect that. Further obfuscations can make detection of cloned functions harder.

### 1.3.3 Ordering obfuscations

The idea behind ordering transformations is to randomize the location of any items in the program - wherever that is possible. The developers tend to organize the code so that logically related things are close together. This works on many levels too, there is locality among statements within BBs, function arguments or functions within the module and so on. For some things, that is easy (functions within the module, arguments of functions), but for changing the placement of statements within BBs, the dependency analysis has to be performed and the options for reordering may be somewhat limited. Potency of these transformations is low, but in many cases, these transformations are one-way.





---

## State-of-the-art

In this chapter, we present some available obfuscation tools. We were interested in obfuscators for compiled languages, especially for languages supported by LLVM (C/C++, Rust, Go, Fortran). We have found that obfuscators for C/C++ are common, for the other languages they are rare. The following sections thus present obfuscators for C/C++ languages, with one exception that is based on LLVM and should thus support the same set of languages.

### 2.1 CXX-OBSUF

CXX-OBFUS<sup>2</sup> is a commercial obfuscator for C/C++. It works on source code level, i.e. takes source code on input and produces obfuscated source code on output. The main feature of this tool is identifier renaming – i.e. it changes identifiers names to a random string of characters. It can consistently rename identifiers in several source files. It also rewrites integer constants into much complicated form (e.g. rewriting 0 to  $0x1fb1+1115-0x240c$ ). This makes the reading of an obfuscated source code harder, but it will not help much against reverse engineering the compiled binary – in fact, optimizing compiler will often simplify this expression again (in constant propagation pass). It seems that this tool does not change the logic of the obfuscated program any further – so it basically performs layout obfuscation.

### 2.2 StarForce C++ Obfuscator

StarForce C/C++ Obfuscator<sup>3</sup> is another commercial obfuscator. We were not able to get this obfuscator and try it, so our analysis is based on their claims and examples of obfuscated code on their websites.

---

<sup>2</sup><http://stunnix.com/prod/cxxo/>

<sup>3</sup><http://www.star-force.com/products/starforce-obfuscator/>

This tool works on source code level too, but unlike the previous tool, it changes the logic of the program, e.g. it adds branches and calls. Examples of obfuscated code show that the code was converted into a virtual machine and the control flow is completely scrambled. It is hard to recognize what other changes were performed on the original, but this tool claims to support over 30 obfuscation methods, including string encryption and insertion of dead code. The user can control the requested level of obfuscation.

Overall, this tool looks quite powerful, but it is closed source, we were not able to try it and they do not provide much detail. Thus, we were not able to explore this tool in a bigger detail.

### 2.3 Tigress C Obfuscator

Tigress[12]<sup>4</sup> is an C diversifier/obfuscator, that started as a research project. It is developed by Collberg (the author of [1]) and others. It works on source code level again. This tool works only with C programming language, it does not support any other language.

It supports many novel obfuscations against both static and dynamic reverse engineering. This tool protects the program by converting it to a virtual machine, with custom instruction sets of arbitrary complexity. It can generate a different virtual instruction set for each function and perform the virtualization multiple times. It also inserts code to make dynamic analysis harder.

This obfuscator seem powerful, featuring many obfuscations that are beyond the scope of this project. This tool does not work automatically, it requires a fine configuration by the user. The user has to specify which functions should be obfuscated and what transformation should be applied to them.

### 2.4 Obfuscator-LLVM

Obfuscator-LLVM[13]<sup>5</sup> is an open-source obfuscator project started in 2010. It features 3 different obfuscation methods: instruction substitution, bogus control flow, and control flow flattening.

Instruction substitution works with integer constants and replaces standard operations with more obscure versions. For example, it changes  $a + b$  into  $a - (-b)$ . Around 10 of these replacement patterns are defined. As stated by the authors, this transformation does not add much potency and resilience (can be easily removed by the optimizer), but it brings diversity into obfuscated binary.

Bogus control flow changes some unconditional branches in control flow graph to conditional. The branch depends on opaque predicate, so one of the

---

<sup>4</sup><http://tigress.cs.arizona.edu/>

<sup>5</sup><https://github.com/obfuscator-llvm/obfuscator/wiki>

branches is never taken (is unreachable), even though it should not be obvious. The unreachable branch is filled with random instructions.

The opaque predicate is not described in the documentation. But after some research of the source code, we found that they add two additional integer global variables ( $x$  and  $y$ ) and whenever they need a predicate, they generate a sequence of instructions to load from that variable and compute the value of  $x * (x + 1) \% 2 == 0$ , which is true for any  $x$ .

Control flow flattening is just as described in the previous section, with optional basic block splitting.

The project, unfortunately, does not seem to be actively developed nowadays. No new feature was added since 2014, there are just occasional changes to make existing code compatible with recent versions of LLVM. Their website mentions project *strong.codes*, a commercial version of this obfuscator, that should support more advanced features. However, at the time of writing this work, the official website of *strong.codes*<sup>6</sup> was not working and we were unable to any additional details about this tool elsewhere.

## 2.5 Summary

As we may have seen in the examples above, the commercial obfuscators usually provide either little or no details about their functionality. While it is logical on one side to keep their effects secret to prevent developing countermeasures, it also makes it hard or impossible to review the quality of performed obfuscations.

Tigress performs many advanced obfuscations, many of them are beyond the scope of this work. However, this tool does not offer an automatic obfuscation of the program, the user must manually configure what should be obfuscated.

Obfuscator-LLVM is quite close to our idea of obfuscator, but that project is not actively developed anymore. We aim to implement more advanced obfuscations. Also, individual obfuscations are independent, so we decided to start our implementation from scratch and not base our code on Obfuscator-LLVM.

---

<sup>6</sup><http://strong.codes/>



---

## Design

In this section, we present the design of our obfuscator. First, we introduce the LLVM framework – with a special focus on LLVM Internal Representation (IR). That is what frames our implementation.

### 3.1 LLVM Framework

In this section, we present the LLVM framework, which we will use as a basis for our implementation. LLVM[14] began as a research project, with aim to provide a modern compilation strategy to support arbitrary programming languages. Nowadays, it is an umbrella project consisting of LLVM core and several other subprojects such as C/C++ frontend *Clang* and various tools and libraries.

LLVM is based on a classical 3-phase compiler architecture shown in figure 3.1. The frontend parses code in a source language and builds a language-specific Abstract Syntax Tree (AST). AST is then converted into a representation that is (source) language and (target) architecture independent –

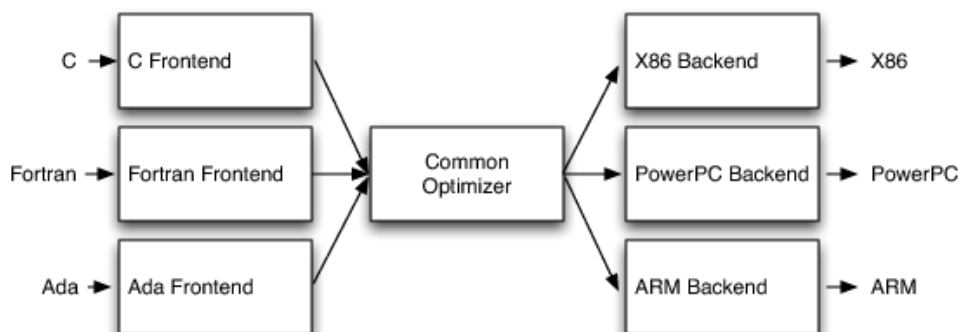


Figure 3.1: 3-phase compiler architecture[14]

```
int factorial(int n){
    if (n==1) return 1;
    return n*factorial(n-1);
}

define i32 @factorial(i32 %n) {
entry:
    %cmp = icmp eq i32 %n, 1
    br i1 %cmp, label %if.then, label %if.end

if.then:
    ret i32 1

if.end:
    %sub = sub nsw i32 %n, 1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul nsw i32 %n, %call
    ret i32 %mul
}
```

(a) (b)

Figure 3.2: LLVM IR example. (a) shows a C function. (b) shows the corresponding function in the LLVM IR form. Labels (*entry*, *if.then* and *if.end*) are basic blocks. Each BB contains a sequence of instructions. Instruction results are stored into virtual registers. The last instruction of a BB is the terminator instruction - i.e., it jumps to another BB or returns from the function.

LLVM IR. LLVM IR aims to be light-weight, yet capable of representing all source languages cleanly. Optimizer performs a variety of transformations on this representation, which usually aims to make code run faster. Finally, the backend takes the intermediate representation and generates binary code for the target architecture.

In the following sections, we focus mainly on the intermediate language of LLVM, LLVM IR. Our implemented obfuscations process this language, so we describe it here, and we also outline some challenges related to this form.

### 3.1.1 LLVM IR

LLVM IR[15] is a source-language-independent representation of input source code. This representation resembles an Instruction Set Architecture (ISA) of a real processor – it defines a set of instructions, and registers<sup>7</sup>. LLVM IR design tries to abstract from machine-specific details such as physical register limitations – it provides an infinite number of virtual registers that can hold values of primitive types (integer, float, and pointer). Figure 3.2 shows a simple example of LLVM IR.

Virtual registers are in the Single Static Assignment (SSA) form. The SSA form requires that each virtual register is written exactly once and it cannot

---

<sup>7</sup>Language reference can be found at <https://llvm.org/docs/LangRef.html>

be changed. This form simplifies data flow analysis, but SSA form presents a problem in case when different values may come from multiple predecessor basic blocks. To solve that, the SSA form introduces *phi* functions.

### 3.1.1.1 PHI function

The SSA form requires that each virtual register is written exactly once, but in most programming languages it is allowed to change values of a variable. In most cases, this does not present a problem, because values are overwritten, and the variable presents the last stored value.

The problem appears when a different value of one variable may come from several basic blocks. Consider the following code snippet:

```
int a;
if (cond()) a=0;
else a=1;
int x=a*2;
```

When variable *a* is referenced on the last line, two different values of *a* may be used, depending on which branch was taken.

To overcome this issue, a special instruction is used in SSA – called the *phi* function. It selects the value depending on where the control flow comes from. This is the corresponding code in LLVM IR:

```
entry:
  %call = call i32 @cond()
  %cmp = icmp eq i32 %call, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %a.0 = phi i32 [ 0, %if.then ], [ 1, %if.else ]
  %mul = mul nsw i32 %a.0, 2
```

The *phi* instruction has the same number of arguments as the number of predecessor blocks. Each predecessor has to define an incoming value. Note that *phi* is an instruction in the LLVM IR form, it does not correspond to any actual machine instruction. This form is used just for analysis.

### 3.1.1.2 Memory operations

All operations in LLVM IR work with values in virtual registers. Virtual registers are not suitable for some uses (e.g., for storing arrays) – thus LLVM IR needs a way to access memory. Memory transfers between memory and a

register are possible via explicit *load* and *store* instructions. If an operation wants to modify a memory location, it has to *load* it into a virtual register, perform the operation, and *store* it back. The operand of a *load* and *store* instruction is either a constant value or a pointer stored in a virtual register.

Virtual registers themselves are not addressable (they behave like *rvalues* in C/C++). All addressable objects (*lvalues*) have to be allocated explicitly. It is possible to define global/static variables – their addresses can be used by memory operations. It is also possible to allocate memory on the stack – by the *alloca* instruction. *Alloca* returns a pointer that can be used by subsequent *load* and *store* instructions. LLVM automatically releases memory allocated by *alloca* on return from the function.

Memory operations actually provide another solution to the problem discussed in the previous section. Instead of using the *phi* instruction, this can be solved by using a local variable:

```
entry:
  %a = alloca i32, align 4
  %call = call i32 @cond()
  %cmp = icmp eq i32 %call, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 0, i32* %a, align 4
  br label %if.end

if.else:
  store i32 1, i32* %a, align 4
  br label %if.end

if.end:
  %a.0 = load i32, i32* %a, align 4
  %mul = mul nsw i32 %a.0, 2
```

In fact, LLVM IR commonly uses both ways. Working with memory decreases the number of inter-block registers (registers that are defined in one BB and used in other BBs). This might be desirable or even necessary for some transformations. This way, it is possible to make all BBs self-contained – in the sense that each BB does not use any virtual registers defined in other BBs<sup>8</sup>. LLVM can also convert between these forms. LLVM refers converting a virtual register into a local (stack) variable as “demoting register to memory” and the opposite process as “promoting memory to register”.

### 3.1.1.3 LLVM API

LLVM offers a C++ API to create custom analysis and transformations. LLVM IR can be represented in 3 different forms: an in-memory data struc-

---

<sup>8</sup>except of stack pointers defined by *alloca* instructions in *entry* block



ture, a binary “bitcode” representation (usually a file with a *.bc* extension) and a human-readable textual form. All these forms are equivalent, and LLVM provides tools to convert between them. LLVM API works with the in-memory form.

LLVM API offers a convenient way to access and modify LLVM IR – each component of LLVM IR (module, functions, basic blocks, etc.) is represented as a class. Member functions can be used to query or modify various information. Classes are polymorphic – e.g., each instruction has a specific class, but all of them inherits from base class *Instruction*. *Instruction* itself inherits from *Value* – instructions represent their result at the same time. Subsequent instructions can use this value.

The in-memory form has various advantages that are exhibited by the LLVM API. Various related entities are linked together – for example, it is possible to iterate over all users of a *Value*. This is convenient on one hand, but care has to be taken to avoid breaking those links when doing various modifications. Fortunately, LLVM API also provides various utility functions to perform more complicated manipulations. For example, *ReplaceInstWithInst* function – as the name suggests – replaces an instruction with another instruction – in fact, that means that the instruction is inserted at the same location, all users of the replaced instructions are changed to use the value of the new instruction, and the old instruction is deleted.

The usual way how the LLVM IR is transformed is by passes run by the optimizer. Each pass performs a specific type of analysis or transformation (such as dead code elimination or loop unrolling<sup>9</sup>). The standard LLVM passes usually tries to make the program faster, and they are automatically run at various optimization levels. The optimizer can also be made to run a custom pass – creating a custom pass is recommended, and a well-documented option<sup>10</sup> how to extend LLVM. We will utilize that option in our implementation.

## 3.2 Implementation design

In this chapter, we present the design of our obfuscator. First, we describe the general way how our obfuscations will be integrated into LLVM. Then, we discuss individual transformations.

We will implement our obfuscator in C++, using the LLVM API (version 4.0.1). We will make our obfuscator work purely as an optimizer pass, without modifying frontends provided by LLVM. Modifying frontends would limit the ability to process an arbitrary language.

---

<sup>9</sup>The list of LLVM passes can be found at <https://llvm.org/docs/Passes.html>.

<sup>10</sup>Documentation can be found at <http://llvm.org/docs/WritingAnLLVMPass.html>.

Each obfuscation will be implemented as a separate optimizer pass. Each pass will be independent of the others, allowing an arbitrary combination of passes.

### 3.2.1 Inlining

Inlining functionality is built-in into LLVM. LLVM automatically uses it<sup>11</sup> with the aim to improve performance – saving the overhead of the call instruction and improving cache locality. It uses heuristics to decide when a function should be inlined – generally, we can say that short functions are more likely to be inlined. Inlining short functions can improve performance significantly, but performance benefit gained by inlining big and complex functions is small. However, our goal is not to improve performance, but to add confusion. This pass will inline more functions than LLVM would have done automatically.

The fact that inlining is already implemented in LLVM makes design of this obfuscation pass significantly easier. LLVM provides a utility function *InlineFunction*, that will modify the CFG for us. This function clones callee into the caller, replaces call by a branch to the clone of the *entry* basic block and replaces all return instructions by a branch to instruction that followed the call instruction. It also takes care of remapping input arguments and returned values.

We could implement these modifications on our own, without using this utility function, but we assume it is much better to use it. There are many potential issues, that need to be handled. For example, if the function is inlined into a loop, all *alloca* instructions have to be moved outside of the loop – otherwise, they would cause stack growth with each loop iteration possibly leading to stack overflow. Also, if the function can throw an exception, handling of that in LLVM IR is not trivial. The utility function is aware of these and many other issues and can handle them properly.

The actual work of this pass will be choosing what to inline. We have to keep in mind that inlining may increase code size significantly. When the function is recursive, it may grow beyond all limits. To prevent that, we will have to set a certain size limit – specified as a certain multiply of original program size. Until the limit is reached, the pass will choose a random call instruction and inline it. The size limit shall be configurable by the user.

### 3.2.2 Split blocks

Several passes will need splitting basic blocks. We decided to implement this as a separate helper pass. A BB can be split into several smaller BBs connected by unconditional branch instruction, as shown in figure 3.3. Splitting blocks itself does not add much obscurity, but it helps the following obfuscations:

---

<sup>11</sup>at certain optimization levels

<pre> 11 : %call1 = call i32 @f1() %call2 = call i32 @f2() %call3 = call i32 @f3() br label %end </pre>	<pre> 11 : %call1 = call i32 @f1() br label %12 12 : %call2 = call i32 @f2() br label %13 13 : %call3 = call i32 @f3() br label %end </pre>
(a)	(b)

Figure 3.3: Splitting basic block example. Both (a) and (b) are equivalent.

- **Bogus control flow:** Increases a number of possible places where a bogus jump can be added.
- **Outlining:** Promotes extracting irrelevant pieces of code (smaller parts of blocks that were previously continuous).
- **Table interpretation:** Increases a number of table states and number of inter-block values.

This pass will randomly choose several splitting points. Each splitting adds one more BB into the function. The target number of BBs shall be configurable by the user.

### 3.2.3 Bogus control flow

Bogus control flow pass will make control flow graphs look more complicated. Unconditional branch instructions are changed into conditional ones using opaque predicates. This seemingly increases a number of possible control paths through the code. Therefore, it increases the perceived code complexity.

This pass will look for unconditional branch instructions at the end of BBs. With a certain probability (configurable by the user), it will replace them by conditional branch instruction. We will implement two options how the bogus control flow can be added. These options correspond to situation (b) and (c) in figure 1.2.

- The conditional branch seems to jump either to the correct successor or that successor's successor (i.e., skipping one BB in the control flow path). An opaque predicate is used to ensure that the correct path is always selected.
- The successor of a BB is cloned and randomly modified, the conditional branch seems to jump either to original or modified BB. An opaque predicate ensure that the original BB is always selected.

Note that we will not insert any opaque predicates in this pass. Instead, we will just insert placeholders (special comparison instructions, that always evaluate to true or false). We will replace them with actual opaque predicates in the Opaque predicate pass, described in section 3.2.7.

### 3.2.4 Outlining

We will use another utility function provided by LLVM to implement outlining. LLVM provides a utility class *CodeExtractor* to do that. *CodeExtractor* is given a code region (a set of BBs, with one marked as the *entry* block) and it performs basically these steps:

1. Determines values needed in the region that are defined outside of the region. These will be the input arguments of the outlined function.
2. Determines values needed outside of the region, that are defined in the region. These will be the output arguments of the outlined function.
3. Demotes register output arguments to stack variables - an address (a pointer) will be passed to the outlined function, the outlined function will store the value there, and it will be loaded before use in the original function. After this step, no output argument remains.
4. Finds all exit BBs from the region (they will stay in the original function).
5. Extracts BBs into a new function and inserts a call instruction into the original function.
6. Inserts an instruction to jump to the exit BBs after the call - if there is just one exit BB, it is unconditional; if there are more exit BBs, it is conditional (in that case, the outlined function returns a value that determines the next BB).

The main work of this pass will, therefore, be to choose which regions to outline. While it is basically possible to choose any call instruction and inline it, it is way harder here. To be able to perform outlining in general, a region must be selected, as we described in 1.3.2.2.

Beyond these formal conditions, we will not outline too simple regions (just one basic block or a a straight line of code). In our opinion, such a simple function would look suspicious to a potential reverse engineer - such simple functions would be most likely automatically inlined during the compilation, so they may realize that the function was extracted by the obfuscator.

We will try to choose the region randomly. We will choose a random *entry* block (among all function blocks), and we will try to choose its successors to get a suitable region. If that will not be successful, selection will be repeated

with another *entry* block. If it will not find a suitable region after several attempts (configurable by user), no further attempts will be performed. The outlining process will be repeated several times on each function, the user shall configure it – either by specifying the maximum number of outlined functions or a maximal reduction of function size (a certain fraction of the original function size).

### 3.2.5 Function interleaving

The function interleaving pass will merge different functions into one. It will choose two functions (with the same return types) and merge them into one. An example of interleaved function CFG is shown in figure 3.4. In essence, this pass will perform these steps to merge two functions:

1. Determine the arguments needed for the new function. One extra argument is needed to distinguish functionalities.
2. Create the function, clone original functions into it and remap their arguments.
3. Insert a conditional branch instruction as the first instruction of the function. It selects the requested functionality based on the value of the extra argument and jumps to the corresponding *entry* BB.
4. Replace all calls to the original functions with a call instruction to the new function.
5. Delete the original functions.

Interleaving functions is easy if both functions have the same arguments. It gets more complicated if they do not. Let us consider the following two functions:

```
void f1 (int a1, int* b1);  
void f2 (float a2, int* b2);
```

```
f1(42, p1); // call to f1  
f2(0.1, p2); // call to f2
```

The easiest way to merge these functions would be to simply join their arguments (arguments in boxes are not needed for the functionality):

```
void interleaved(int a1, int* b1, float a2, int* b2, bool func);
```

```
interleaved(42, p1, 0, NULL, true); // call to f1  
interleaved(0, NULL, 0.1, p2, false); // call to f2
```

However, that is not optimal, we will try to improve that by reusing arguments. Both original functions have one *int\** argument in common. We want the following result:

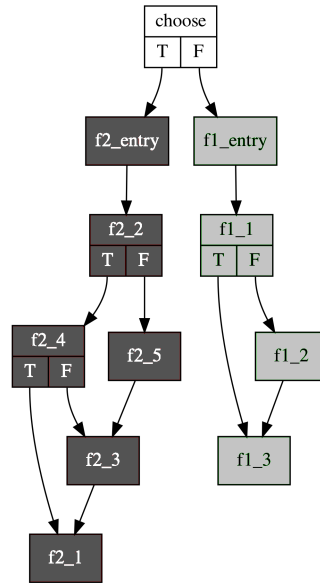


Figure 3.4: Interleaved function CFG example. BBs with a different color were in different function before – they have different functionalities. The *entry* BB selects the functionality and jumps to the requested functionality. This can become more complicated when several functions are merged into one.

```
void interleaved(int a1, int* b, float a2, bool func);

interleaved(42, p1, [0], true); // call to f1
interleaved([0], p2, 0.1, false); // call to f2
```

We can see that this saved one argument. In both calls, there is still one useless argument, but as they are of different types, we cannot simply reuse them. At least, we will use them to add some more confusion – we will fill these useless arguments with random values. Also, we want the functionality distinguishing argument to be at an arbitrary place between other arguments (not always the last one). In addition, the argument distinguishing functionalities can be protected by an opaque predicate.

We will try to choose functions with similar arguments for interleaving. We will choose several functions randomly, and we will try to find a function with similar arguments to interleave with. The number of interleaved functions shall be configurable by the user. Also, the user shall be able to control the maximum number of functionalities contained within one function.

### 3.2.6 Table interpretation

This pass will break the original control flow completely, and it will make it hard to track which block follows which. All BBs (resp. their addresses) will be stored in a table, and they will be assigned a number (their index in the table). BBs will not jump to their successor; they will jump to a special BB (called *dispatcher*). Before jumping there, they will store an index of their successor to a special *state* variable. The *dispatcher* will load that value and will jump to the corresponding entry in the table.

It is easy to implement this for blocks with a single successor - we will store a single value into *state* and jump to *dispatcher*. However, when a BB has more successors, it branches to several BBs, depending on a condition. We will use the *Select* instruction to solve this situation. That instruction chooses a single value depending on the condition without branching (this corresponds to a conditional move in some CPU architectures). Thus, we can also replace a conditional branch with an unconditional one. Figure 3.5 shows an example.

#### 3.2.6.1 Hardening

Chow et al.[10] suggest that a way to make an analysis of flattened CFG harder is to expand the number of states (BBs). We will use that suggestion to improve our implementation. They propose to split BBs into smaller pieces – we do this in the Split Blocks pass. Another idea is to add dummy states. We will utilize this idea here too – we will clone the original BBs of a function – and we will let them jump to arbitrary BBs. We will also add BBs from another functions. These BBs will be unreachable, but they will increase information load for a potential reverse engineer. This increases the state space considerably, but without additional modifications, it would be possible to track possible values of *state* variable to detect states that are reachable. We would like to prevent that, so we will implement additional modifications.

We will try to make the branches to *dispatcher* more obscure. These modifications are similar to what Bogus Control Flow does, but some additional things are possible here, that were not possible before table interpretation (e.g., replacing the conditional branch with an unconditional one was not easily possible). Recall that for table interpretation (without any modification) we would:

- Replace an unconditional branch to a successor with an unconditional branch to the *dispatcher* and store the successor's number into *state* before that.
- Replace an conditional branch to two successors with an unconditional branch to the *dispatcher* and add the *select* instruction to choose number of the successor (and storing it to *state*).

	<pre>new_entry:   %state = alloca i32   store i32 0, i32* %state   br label %dispatcher</pre>
	<pre>dispatcher:   %1 = load i32, i32* %state   switch i32 %1, label %def [     i32 0, label %entry     i32 1, label %l1     i32 2, label %l2   ]</pre>
<pre>entry:   %1 = call i32 @f1()   br %1, label %l1, label %l2</pre>	
<pre>l1:   call i32 @f2()   br label %l3</pre>	<pre>entry:   %2 = call i32 @f1()   %3 = select %2, i32 1, i32 2   store i32 %3, i32* %state   br label %dispatcher</pre>
<pre>l2:   ret i32 0</pre>	<pre>l1:   call i32 @f2()   store i32 2, i32* %1   br label %dispatcher</pre>
	<pre>l2:   ret i32 0</pre>
	<pre>def:   unreachable</pre>
(a)	(b)

Figure 3.5: Table interpretation - LLVM IR example. (a) shows the original code and (b) shows the code after table interpretation obfuscation. All branches have been replaced by a branch to *dispatcher*. The conditional branch in *entry* has been replaced by an unconditional branch and the *select* instruction has been added.



The reachable BBs never jump to a cloned (unreachable) block. We will use opaque predicates to make them look like they do. There are several options how to do it:

- Replace an unconditional branch with a conditional branch to the *dispatcher* and an arbitrary BB. Store the number of the successor into *state* and always jump to the *dispatcher*.
- Replace an unconditional branch to a successor with an unconditional branch to the *dispatcher*. Add a *select* instruction, always choose the value of the successor and store it into *state*. The other value in *select* may be a number of an arbitrary BB.
- Replace a conditional branch to two successors with a conditional branch to the *dispatcher* and one successor. Store the number of the other successor into *state*.

These modifications will rely on opaque predicates. They will make the analysis even harder – if the opaque predicates will not be detected, one will not be able to identify unreachable BBs just by tracking the *state* value (reachable blocks may also seem to jump to them).

Furthermore, we will use one idea from [16]. We will not store the number of the successor BB as an absolute value at the end of each BB. Instead, we will store the difference between the number of current BB and the number of its successor (except of the first store in *entry* BB, it will have to store an absolute value). This may make it even more difficult for a potential reverse engineer – they will not be able to determine the successors of any BB, they will need to start from the *entry* block and track the value of *state* variable. We will have to be careful not to interfere with the previous described modification though.

### 3.2.7 Opaque predicates pass

Some passes described earlier use opaque predicates. These passes will insert placeholders instead of opaque predicates. These placeholders are functional (we will use a special compare instruction that always evaluates to true or false), but they are not resilient. This pass will replace those placeholders with real opaque predicates that should be more resilient.

First, this pass will insert several integer global variables. Then, it will look for placeholder instructions. It will replace them with a sequence of instruction to load from one of those global variables and use that value to evaluate an invariant expression (from table 1.1). Furthermore, it will also modify the value of that global variable – it will either modify it by a random number (fixed at compile time), or it will modify it by a random function argument (if any is present). Since the expressions are invariant (they evaluate always to

the same value, regardless of input variable values), the value of those variables is not important – we randomly update them just to increase confusion. We will use atomic instructions to load and store values, to ensure correct behavior in multi-threaded environment.

#### 3.2.7.1 Resilience

We try to evaluate resilience of such opaque predicate design. A potential deobfuscator cannot simply evaluate the expression, since it does not know the value of the input variable. If it would try to keep track of the global variable value, it would need inter-procedural analysis – the value is updated from different functions and by values that are passed between these functions. Note also that the values of function arguments might depend on user input (e.g., *argc* – the number of program arguments in the *main* function). Thus, we assume that a deobfuscator might not be able to track the value of global variable just by analysing the program itself – it would also need to know possible program inputs.

However, some approaches for identifying opaque predicates[6, 7] do not rely on tracking possible input value. Instead, they rely on constraint solvers to find out if an expression is invariant. These tools would be able to identify our opaque predicates. By reviewing the procedure of such tools, we found out that their procedure still requires a non trivial work, even though it is probably simpler then the previous approach. We have concluded that this approach would need to analyse the whole program and test all expression whether they are invariant – this will render our predicates as having *weak* resilience.

We conclude that our predicates will have *weak* resilience (if a constraint solver would be used to prove that our expressions are invariant). If a deobfuscator would try to track values contributing to opaque expression, our predicates will have at least *strong* resilience.

#### 3.2.8 Removing identifier names

For identifier renaming, we will use *StripSymbol* pass, that is built-in into LLVM. This pass removes all symbols – debug symbols and global variables and functions names. Note that it is not allowed to remove some names – they are needed for program linking. We will discuss this limitation later in more detail.

---

# Implementation

In this chapter, we describe details of our implementation based on the design from the previous parts. In the text, we also discuss some practical issues we encountered. At the end of this chapter, we describe how our obfuscator can be used and the limitations resulting from that.

Our primary concern when implementing obfuscation transformations was preserving the behavior of obfuscated programs. To ensure that, we had safeguards on various levels. First, we run the verifier pass after our obfuscation passes. That pass checks if LLVM IR is well-formed, e.g. if each BB has a terminator instruction or if each use of a value is dominated by its definition. We have many various checks in the obfuscation passes itself, that try to make sure that the obfuscation can be performed. Finally, we also used several test programs, we ran them through our obfuscator and checked if the results of obfuscated and non obfuscated versions are the same.

## 4.1 Implementation details

### 4.1.1 Inlining pass

Implementing the inlining pass was straightforward, and we implemented it just as planned. The only issues we had was with removing unused functions. Most importantly, we realized it is only possible to remove functions that are local to the module (this corresponds to C/C++ *static* keyword - LLVM denotes this linkage as *internal* or *private*). Functions with *common* linkage shall not be removed (not even renamed), as other modules might be using them (and at the compilation stage, we do not have information from linking stage). We discuss this limitation in more detail in section 4.2.1.

At first, we tried removing functions manually in this pass, but we encountered some problems with metadata. Some functions were referenced by LLVM metadata and removing them produced broken LLVM IR. We solved these issues by using built-in Global Dead Code Elimination pass for removing

unused functions. This pass only removes functions that are legal to remove and also handles removing of linked metadata.

#### 4.1.2 Bogus control flow pass

Implementing the bogus control flow pass was challenging due to the SSA form of LLVM IR. When we add a new predecessor to a BB, we also need to add *phi* instructions to collect used virtual registers coming from that predecessor. If there already were some *phi* instructions, they have to be modified for the new number of predecessors. We encountered many problems when we tried to fix *phi* instructions ourselves. We solved this problem by “demoting” inter-block registers (registers used outside of the block where they were defined) to stack variables, performing the requested modifications and “promoting” them back. LLVM provides utility functions to do that, and we found out that it is much easier to work with this form for the modification we do here.

#### 4.1.3 Function outlining pass

We have designed the outlining pass to use LLVM utility functionality *CodeExtractor* to extract a code region. We found out that *CodeExtractor* has a bug that sometimes results in broken IR. If there were *phi* instructions in exit BBs from the region, this function modified them incorrectly. We were not able to find out the exact cause of that bug, but we managed to work around this issue by “demoting” all *phi* instructions in exit blocks to stack variables before performing the extraction. This workaround seemed to solve this problem – after applying it, we have no further problem with this function.

#### 4.1.4 Table interpretation pass

The table interpretation pass hides the actual flow of control – doing so also destroys the dominance relation mandated by the SSA form of LLVM IR. All inter-block virtual registers have to be converted to memory loads and stores. This conversion is possible thanks to the fact that terminator instruction does not return a value, so it is possible to insert a *store* instruction before terminator instruction. There is one exception, however - and that is the *invoke* instruction, which is used for calling functions that may throw exceptions.

The *invoke* instruction terminates a BB – program execution may continue to normal destination or to unwind destination (when an exception happens). It also returns a value – a return value of the called function. Thus, it is not possible to store this value before jumping to another BB. We handle that by not changing the destination of *invoke* and storing its result in its successor BBs. Thus, after an *invoke* instruction, the flow of control does not jump to *dispatcher* (as it does after any other BB), but continues to the actual successor - and that jumps to *dispatcher* again. That way, we can apply table interpretation to functions having *invoke* instructions as well.

In example 3.5 we used a *switch* instruction in the *dispatcher* blocks. In our implementation, we use an *indirect branch* instruction. The *switch* instruction is easier to use, with the *indirect branch* instruction we need to take care of creating the jump table ourselves. However, it allows us to make it more obscure – we can, for example, add references to BBs from another functions or invalid entries into the jump table. That confuses software used for reverse engineering – we will see an example in the next chapter. On the other hand, using indirect branch has one disadvantage – the function cannot be cloned after that. Thus, it is not possible to inline or interleave function after table interpretation.

#### 4.1.5 Metrics pass

We have created a separate pass for computing code metrics. We will use this for evaluating obfuscations in the next chapter. Implementing this as a pass has the advantage that we can insert this pass in between any other passes, and we can track how the metrics change during the obfuscation process.

We implement code complexity metrics from section 1.2.1 - code size ( $\mu_1$ ) and computational complexity ( $\mu_2$ ). We compute  $\mu_1$  by adding up the number of instructions and the number of their arguments. In other words, each instruction contributes to  $\mu_1$  by  $1 + argNo$ . To compute  $\mu_2$ , we first count nodes (BBs) and edges of CFG and then we compute  $\mu_2$  by using formula[17]  $edges - nodes + 2$ . This pass computes these metrics for the whole program and also for individual functions - and reports the maximum and the average of their values.

#### 4.1.6 Scheduler pass

We also implement one pass to ease scheduling of obfuscation passes. This pass runs all the previously listed passes and optionally prints metrics.

The passes are run in this particular order:

1. Inlining pass
2. Splitting basic blocks
3. Bogus control flow
4. Outlining
5. Function interleaving
6. Table interpretation
7. Opaque predicates adding
8. Stripping identifier

The rationale behind the order is the following: inlining, splitting basic blocks, and bogus control flow all increases size of a function. Longer function means a better chance for outlining. Splitting basic blocks is useful for the bogus control flow pass, outlining and table interpretation, so it should be scheduled before them. Function interleaving can then interleave functions that were outlined. Table interpretation must come after all these, because (as discussed in 4.1.4), functions cannot be copied after this pass – inlining or interleaving cannot process that function anymore. Finally, we schedule the opaque predicates pass – after all passes that use opaque predicates. We remove any names at the end.

The user may choose to change the order of passes – arbitrary order is possible. The only limit known to us is that inlining and function interleaving will not work after table interpretation.

## 4.2 Usage and limitations

All described passes form a library and they are compiled to one shared object file (*LLVMObfusculator.so*). LLVM tools can load this file to run obfuscations. The basic usage that should work for an arbitrary language (frontend) is the following:

1. Use the language-specific frontend to obtain LLVM IR.
2. Use LLVM optimizer tool (*opt*) to run obfuscation passes.
3. Use linker (LLVM *lld* or language-specific) to link LLVM IR into an executable file.

For example, for C++ this in practice means the following commands:

```
clang++ -c -S -emit-llvm -o source.ll source.cpp
opt -S -load LLVMObfusculator.so -obfuscator -o source_obf.ll < source.ll
clang++ source_obf.ll
```

Some frontends (clang, clang++) allow doing that just by one command:

```
clang++ -Xclang -load -Xclang LLVMObfusculator.so source.cpp
```

This option allows easy integration of our obfuscator into existing compilation workflow – just by passing several more flags to the compiler. However, not all frontends support this option, the first method should work for any frontend. More elegant ways can be created by modifying the specific frontend – but we decided not to do that to keep our obfuscator language-independent.

Most of the passes offer some configuration options. Running the obfuscator without any arguments uses the default configuration. The way to override the default settings is to set an environment variable. Overview of user-tunable parameter and their default values can be found in Appendix B.

The described method works just with one module at a time – eg., functions are inlined or interleaved within one module. If the program consists of several source code files, that are compiled separately and linked together into an executable, it may be useful to allow mixing functions from different modules. To do that, individual modules need to be merged into a single LLVM IR. LLVM tool *llvm-link* can be used for that:

```
clang++ -c -S -emit-llvm -o a.ll a.cpp
clang++ -c -S -emit-llvm -o b.ll b.cpp
clang++ -c -S -emit-llvm -o c.ll c.cpp
llvm-link -S -o comb.ll a.ll b.ll c.ll
opt -S -load LLVMObfuscator.so -obfuscator -o source_obf.ll < comb.ll
clang++ source_obf.ll
```

This, however, does not solve the limitation that we are not able to remove some functions and/or their names. If functions have *common* linkage, they cannot be removed, because it is not clear if they will be needed in the linking process. Our obfuscator has no information from the linking process.

#### 4.2.1 No link-time informations

LLVM does not provide any link-time information at the stage when our obfuscator runs – after the compilation stage and before the linking process starts. Lack of link-time information limits our ability to remove unused functions and identifiers – functions with *common* linkage might be used by other modules, so they must be kept. A function name might help a potential reverse engineer to quickly understand the purpose of that function. Left function prototypes (of functions that were inlined) might help in recognizing which part of function was previously in a separate function.

We see three possible ways to solve this: using link-time optimization, declaring functions as internal or using external tools to strip names.

LLVM offers link-time optimization that can remove dead function prototypes. It works as a plugin into the system linker<sup>12</sup>. This cannot be done automatically by our obfuscator, as it requires an additional configuration by the user. Thus, it is a possible solution, but it is outside of the scope of this project.

Users can improve obfuscation by declaring functions as having internal linkage (*static* keyword in C/C++) when they are not needed from another module. That allows to remove some function prototypes and/or their names, but not all of them, as some functions are likely needed by other modules.

Another solution is to use an external tool to strip symbols – for example *strip* tool from GNU binutils<sup>13</sup>. The disadvantage of this approach is that it only removes function names, not function prototypes.

---

<sup>12</sup><https://llvm.org/docs/GoldPlugin.html>

<sup>13</sup><https://www.gnu.org/software/binutils/>





---

# Evaluation

In the previous chapters, we described obfuscations and our implementation of obfuscator. In this chapter, we evaluate obfuscation metrics – potency, resilience and performance impact. We conclude this chapter with a comparison of our obfuscator with existing tools.

## 5.1 Obfuscation metrics

In [1], obfuscations we have implemented are described as having the following properties:

Table 5.1: Suggested obfuscation properties

Obfuscation	Potency	Resilience	Cost
Inlining	medium	one-way	free
Bogus Control Flow	depends on opaque predicate		
Outlining	medium	strong	free
Function interleaving	depends on opaque predicate		
Table interpretation	high	strong	costly
Removing identifiers	medium	one-way	free

Note, however, that their implementation details were not described in that article, so the evaluation might not hold.

### 5.1.1 Potency

In this section, we evaluate potency of obfuscations. Article [1] defines potency as a measure of difficulty to understand the obfuscated program compared to the not obfuscated one (for a human). It uses a three-point scale (low, medium, high) for measuring potency. It furthermore suggests using software complexity metrics to evaluate potency. However, the article does not say how these

Table 5.2: Program size changes ( $\mu_1$ ) for test programs. Values indicate ratio between metrics of obfuscated and non obfuscated program. Total metrics refers to the whole program, max is the maximum value among functions. (TI – table interpretation, BCF – bogus control flow)

	AES		QuickSort		MatrixMult	
	total	max	total	max	total	max
all (default)	17.37	35.37	25.34	25.56	6.94	7.83
all, no splitting	13.27	32.73	14.93	15.13	3.94	4.28
all but TI	7.73	17.87	8.85	7.90	2.72	2.30
all but TI, no splitting	6.28	11.14	6.83	5.75	1.83	1.60
inlining, size limit 2	2.24	6.13	2.26	2.84	1.13	1.78
inlining, size limit 4	5.39	17.91	4.33	4.72	1.13	1.78
inlining, size limit 8	6.20	20.54	8.35	11.70	1.13	1.78
BCF prob 0.5	1.32	1.57	1.49	1.01	2.14	1.96
BCF prob 0.5, no splitting	1.21	1.05	1.17	1.00	1.46	1.41
outlining	1.08	1.01	1.10	1.00	1.37	0.91
outlining, no splitting	1.06	1.00	1.01	1.00	1.20	0.62
interleaving	1.33	3.82	1.05	1.04	1.00	1.00
TI	2.56	2.64	2.94	1.94	3.86	3.82
TI, no splitting	2.15	2.31	2.45	1.98	2.73	2.69

metrics relate to the proposed scale for potency. We measure these metrics for several test programs and then we discuss how implemented obfuscation makes programs more confusing for a potential reverse engineer.

#### 5.1.1.1 Software complexity metrics

We tested programs written in various programming languages – MatrixMult is in C++, AES in C and QuickSort in Rust. These programs are attached to this thesis. We tested them with various obfuscation parameters and also individual obfuscations. Table 5.2 shows changes in code size – the size of the whole program and also the size of the largest function. Table 5.3 shows changes in computational complexity – for the whole program and also the complexity of the most complex function. Unless mentioned in that tables, obfuscation uses the default configuration, as described in Appendix B.

Inlining increases both metrics; its results vary depending on the nature of the program. For AES and QuickSort program, the increase was significant. In those programs, there are many possibilities for inlining – AES program contains functions that can be inlined into multiple locations, QuickSort program has a recursive function, that can be inlined into itself without limits. We can also notice that the effect of inlining was even more significant when size limit was increased. In case of MatrixMult program, the metrics change

Table 5.3: Computational complexity changes ( $\mu_2$ ) for test programs. Values indicate ratio between metrics of obfuscated and non obfuscated program. Total metrics refers to the whole program, max is the maximum value among functions. (TI – table interpretation, BCF – bogus control flow)

	AES		QuickSort		MatrixMult	
	total	max	total	max	total	max
all (default)	11.36	19.35	29.96	63.38	5.13	6.86
all, no splitting	6.12	9.60	17.50	41.12	3.10	3.54
all but TI	3.24	5.80	5.41	14.00	1.61	2.00
all but TI, no splitting	2.54	5.25	4.65	9.12	1.22	1.25
inlining, size limit 2	1.36	2.20	1.78	3.50	0.97	1.46
inlining, size limit 4	2.04	4.70	3.19	7.25	0.97	1.46
inlining, size limit 8	2.03	4.70	6.22	22.25	0.97	1.46
BCF prob 0.5	1.43	2.00	1.33	1.88	1.57	1.89
BCF prob 0.5, no splitting	1.17	1.40	1.00	1.00	1.20	1.32
outlining	1.13	1.00	1.02	1.00	1.17	0.46
outlining, no splitting	1.13	1.00	1.00	1.00	1.14	0.50
interleaving	1.26	1.50	0.96	2.00	1.00	1.00
TI	3.74	5.75	3.87	7.88	4.17	5.86
TI, no splitting	2.18	3.25	2.02	3.88	2.35	3.14

was less significant. In that program, there are just two functions that can be inlined. The growth can be observed mainly on the maximum of those metrics, values for the whole program were almost changed – that is because function bodies were “moved” into their calling function and then deleted. We can notice that the  $\mu_2$  metrics for the whole program slightly decreased – because computation complexity of the whole program also depends on the number of functions in the program.

The main metrics that bogus control flow affects is cyclomatic complexity ( $\mu_2$ ). We can observe, that when it was used without BBs splitting, it did not affect the metrics very much. Splitting BBs created more locations where bogus flow could be added. We can notice that with splitting blocks enabled, bogus control flow affected the metrics more significantly. Still, its effects are quite low compared to inlining.

Outlining influenced only one program – MatrixMult. We can see that while the total of both metrics slightly increased, the maximum value in fact decreased. This fact is caused by the way outlining works – it extracts some code (and complexity) to a separate function. Thus, we have to conclude that those metrics are not suitable for evaluating outlining.

Interleaving only affected AES and QuickSort programs. However, the effects are different. In case of QuickSort program, it interleaved functions and removed them – thus the total values of both metrics did not change,

but the maximum values increased. In case of AES, it was not possible to remove merged functions as they had *common* linkage. Thus, the total values of metrics increased too – the code was copied into the merged function. Overall, metrics change caused by interleaving is small.

Finally, table interpretation caused the most significant growth of both metrics for all programs. Unlike the previous obfuscations, this obfuscation does not seem to be very sensitive to the program’s nature – the results are similar for all programs. We can also notice that splitting BBs helped this pass significantly.

Overall, we can conclude that obfuscations are successful at increasing the selected metrics. Moreover, we see that the results do not depend on the programming language. They depend on the nature of the obfuscated program, though. Some obfuscations themselves do not affect some of the test programs, but when they were used together, they managed to increase metrics of each obfuscated program.

#### 5.1.1.2 Confusion for a reverse engineer

We have seen the way obfuscations affect software complexity metrics in the previous section. However, we do not know how the metrics changes relate to potency. The article [1] does not say that. Potency describes the amount of confusion added by the obfuscations for a potential reverse engineer. To get more insight into that, we try to discuss that here.

Inlining obfuscation causes a significant code growth. A potential reverse engineer would need to study a lot more code after inlining obfuscation. As discussed in 1.3.2, functions provide an abstraction for developers. This abstraction is removed by inlining.

Bogus control flow obfuscation adds unreachable BBs into the program. A potential reverse engineer would see that as a branch instruction depending on opaque predicate. Thus, the confusion of bogus control flow depends on whether a reverse engineer would realize that the predicate is invariant. We use global variables and invariant expressions to implement opaque predicates. Even though we use several expressions to implement opaque predicates, all opaque predicates are similar – they load a value from a global variable, perform some arithmetics and convert the result to bool value. The value is then used as a predicate for a conditional jump instruction. Thus, all our opaque predicates have a similar signature and a potential reverse engineer may realise that. This could be possibly improved by a bigger variety of opaque predicates (not only the expressions used).

Outlining is the opposite of inlining. It extracts some code from a function into a new function and thus creates a bogus abstraction. A common reason why developers create functions is to isolate some functionality – functions perform one specific task. However, this is not true for functions created by outlining obfuscation. The confusion caused by this depends on whether

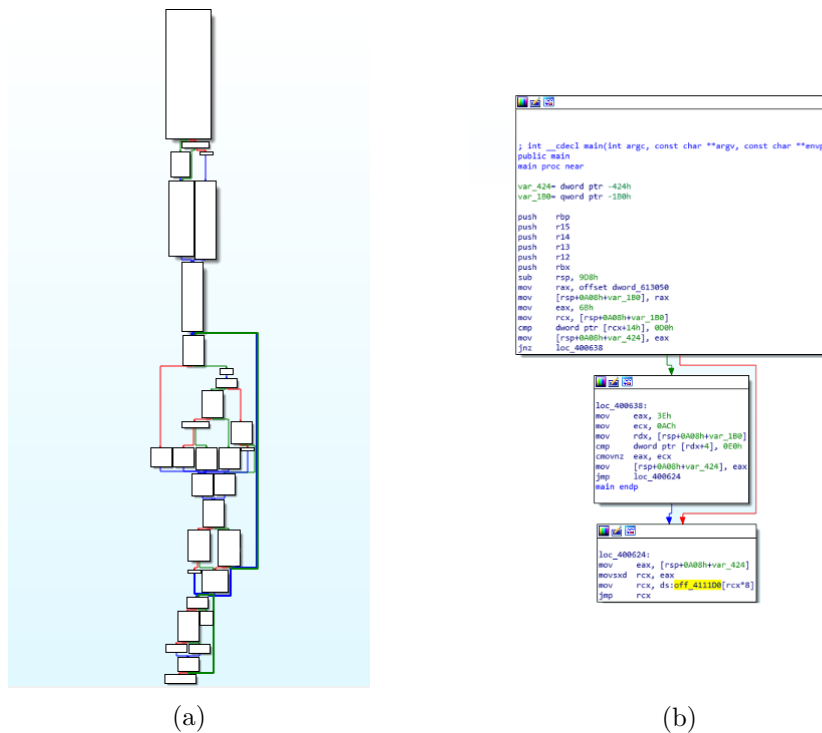


Figure 5.1: Visualization of table interpretation effect in IDA (version 7.0). (a) show the original CFG, (b) shows the resulting CFG after table interpretation. IDA is not able to visualize the real CFG after table interpretation.

a reverse engineer would realize that an obfuscator outlined the function. In our implementation, one clue for them might be that all outlined functions are called from just one location. However, when outlining is combined with interleaving, outlined functions might be called from several locations.

Interleaving creates the confusion by merging several functions into a single one. Confusion again depends on whether a reverse engineer notices that fact. They would need to explore instructions calling that function and the structure of the function itself. One possible weakness of our implementation is that functions are merged in a simple way – the first branch of the function depends on one argument which selects the functionality. The control flows of different functionalities never merge. This might indicate an interleaved function. We can possibly improve that by merging functions in a more complicated way – e.g., by using multiple arguments and branches to select the functionality.

The confusion caused by table interpretations lies mainly in the fact that it completely hides the CFG of a function. By exploring the CFG, a reverse engineer may quickly get an idea how complex is the function is and how it works – e.g., if there are any loops, where the branches in the function are or what functions are called by this function. After table interpretation, they

are not able to do that anymore. To illustrate that, we show a visualization of a CFG of a function in IDA (version 7.0), a popular reverse engineering tool in figure 5.1. A reverse engineer might try to track the values of a variable determining the next destination, but that is more demanding, and our implementation makes it even more complicated, as described in 3.2.6.1.

Function names are useful for reverse engineers because they allow them to guess a purpose of a function quickly. This may allow them to quickly skip utility functions and focus on the relevant pieces of code. By removing names, we prevent the distinction between relevant and irrelevant pieces of code.

### 5.1.1.3 Summary

We have evaluated software complexity suggested in [1] to measure potency. We have furthermore discussed the confusion that our obfuscator creates for a potential reverse engineer. We conclude this section by reviewing whether the potency evaluation suggested holds for our implementation.

Inlining is described as having medium potency. Based on the significant increase of metrics, we say that it has medium potency in our implementation as well.

Bogus control flow does not have potency specified; it is said that it depends on the opaque predicate. Based on our evaluation that the opaque predicate is not very potent and also on the fact that bogus control flow barely increased the metrics, we say it has low potency.

Outlining is described as having medium potency. However, we have observed that the used metrics are not suitable for its evaluation. Based on the previous discussion, we assume that potency of outlining itself is low. We can increase potency by using outlining in conjunction with interleaving.

Function interleaving is said to depend on opaque predicates. In our implementation, we do not always use opaque predicate to protect it. Based on the low increase of metrics, we would say that interleaving has low potency. Nevertheless, we discussed that it might cause significant confusion for a potential reverse engineer.

Table interpretation is said to have high potency, but the metrics increase we observed was comparable to inlining with medium potency. Thus, we assume that the potency of table interpretation we implemented is medium as well. We suppose that in [1], they meant a more complex form of table interpretation.

Removing identifiers is described as having medium potency. Based on the previous discussion, we agree with that.

### 5.1.2 Resilience

Resilience measures the difficulty of creating an automatic deobfuscator. We are not aware of any generic deobfuscation tool. Creating a deobfuscator

would be an entire project of its own, a potential reverse engineer would need to design a deobfuscator for each specific obfuscation. We discuss the aspects of each obfuscation in terms of writing a deobfuscator.

A major challenge for a deobfuscator would be opaque predicates. We have discussed resilience of used opaque predicates in section 3.2.7.1. We have concluded that implemented opaque predicates have weak resilience – if an approach from [6] is used. If all opaque predicates were identified and removed, it would be possible to simplify conditions and remove all unreachable BBs.

Removing inlining obfuscation is not possible – it is one-way. Note that this is true only if the function prototype is removed after inlining. We discussed in 4.2.1 that in some cases it is not possible to remove the function prototype. In that case, a deobfuscator might be able to figure out that the function has been inlined.

Resilience of bogus control flow obfuscation is determined by the resilience of opaque predicates. If opaque predicates are identified, a deobfuscator would be able to find unreachable branches and remove them. Opaque predicates we use have weak resilience; thus resilience of bogus control flow is weak too.

It is always possible to inline outlined function, thus it is also possible to automatically remove outlining. The problem is that the deobfuscator might not recognize whether the function has been created by the developer (so it presents a useful abstraction and should be kept) or whether it is bogus (and thus should be inlined). In [1], they described outlining as having strong resilience.

To remove interleaving, a deobfuscator would need to find out if a function could be split into several functions. For that, it would have to analyze arguments of all call instructions calling that function. Furthermore, it would need to consider whether the function has been created this way by a developer or by an obfuscator, a similar problem as described with outlining. Thus, we assume that resilience is strong.

Resilience of table interpretation in our implementation depends on opaque predicates as well. After removing opaque predicates, a deobfuscator would be able to track the values of variable determining next BB. That would allow the deobfuscator to reconstruct the CFG. Thus, table interpretation has *weak* resilience.

Finally, resilience of identifier removing is one-way. Once the names are removed, it is not possible to recover them. Note that it is not possible to remove some names due to the limitation described in 4.2.1.

We have evaluated resilience of implemented obfuscations. Our evaluation mostly agrees with evaluation in [1] – with the exception of table interpretation. Similarly, as in the evaluation of potency, we see that our implementation of table interpretation is simpler than the one described there. We evaluated resilience of our implementation as having weak resilience.

### 5.1.3 Performance impact

In this section, we evaluate the performance impact of obfuscations. We have noticed that the performance of obfuscated program may significantly vary. We explain the reason for that in the following part and then we present the measured performance impact.

#### 5.1.3.1 Variance in results

Obfuscations are randomized, thus they may affect input program in various ways. We have also found out that the performance of obfuscated program varies. The same program, with the same obfuscation configuration, was in some cases significantly slower than usual. Such behavior is undesirable, we tried to find out why this happens.

Let us take for example one of our test programs, the matrix multiplication program. The core of that program consists of three nested loops. Most processor time is spent in the inner loops – the code in the inner loops is “hot”. We hypothesized that the performance impact is dependent on the level of obfuscation of these code parts.

We made a series of experiments to prove that and we found out that significant performance impact is caused already by the split basic blocks pass. This pass randomly chooses several splitting points and splits the BBs. We found out that this itself is responsible for a large variance in performance.

When BBs in the inner loops were split into many pieces, the performance of program decreased significantly. The code in the “hot” zone was very compact, and even the unnecessary jump instructions made it significantly slower. We have then modified splitting basic blocks to lower the chance that inner loops are split (we made the probability of splitting to decrease exponentially with nesting level). That helped significantly with the performance – as we can see in figure 5.2.

A similar issue also happens in outlining pass – outlining in inner loops causes a significant slowdown. When the outlining region was chosen randomly, the results varied significantly. We have added a similar condition as we did in splitting blocks pass.

This way, we have reduced obfuscations performed in the inner loops – and in turn improved performance. We can see the impact of these changes in figure 5.3. Note that while we managed to reduce slowdown caused by obfuscating inner loops, this does not completely solve the problem of obfuscating “hot” code zones. A better solution might be to let the user mark the code parts that should be less obfuscated.

#### 5.1.3.2 Performance impact on test programs

We prepared several testing configurations for obfuscation and then we evaluated our test programs. All test programs were compiled with the highest



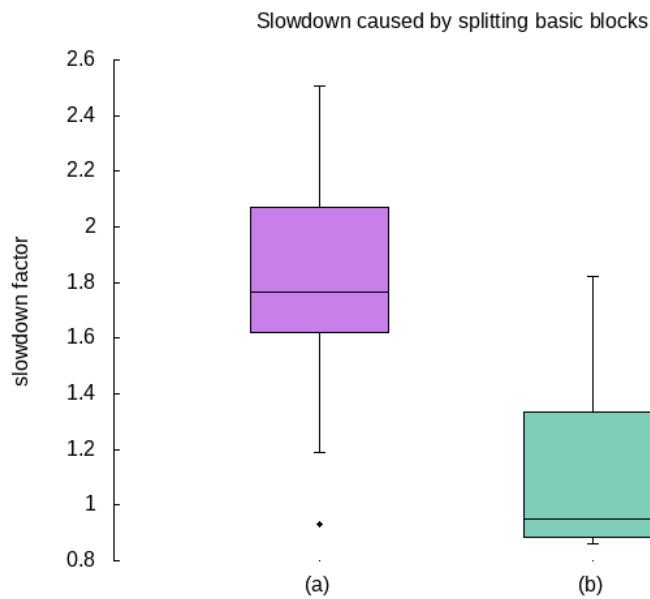


Figure 5.2: Slowdown caused just by splitting basic blocks (split factor 2). (a) shows results when the splitting points are selected uniformly randomly. (b) shows results when blocks with lower nesting level are preferred for splitting.

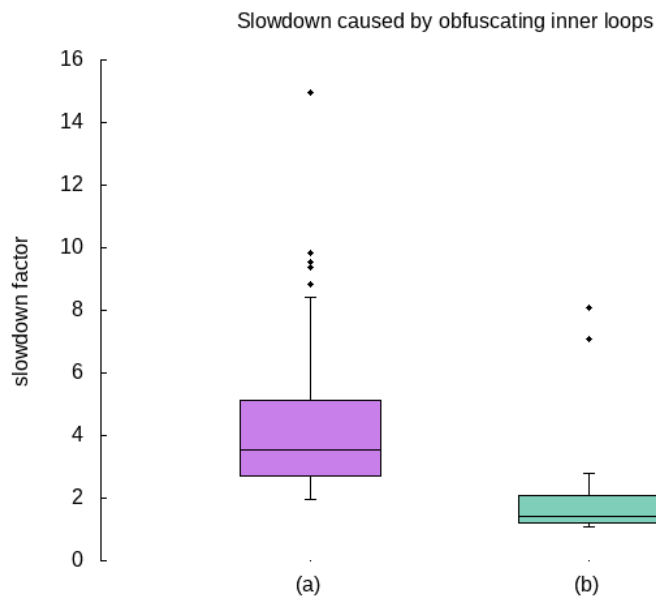


Figure 5.3: Slowdown factor with all obfuscations enabled. (a) shows results when the splitting points and outlined regions are selected uniformly randomly. (b) shows results when blocks with lower nesting levels are preferred for splitting and outlining.

optimization level ( $-O3$ ) and then obfuscated. We compare the performance of obfuscated program with the performance of non obfuscated program.

In figure 5.4 we can see the resulting time for matrix multiplication program (C++), a  $O(n^3)$  algorithm. In figure 5.5 we can see the results of various obfuscation levels related to time of non obfuscated program. It seems that the performance impact caused by obfuscation does not grow with the matrix dimension.

In figure 5.6 we can see the resulting time for merge sort program (C++). Computational complexity of this algorithm is  $O(n.\log(n))$ . In figure 5.7 we can see the results of various obfuscation levels related to time of non obfuscated program. Our conclusion is the same – the obfuscation overhead does not grow with problem size.

Finally, we made a test with AES program (C). This program performs repeated encryptions using AES – the complexity grows linearly with the number of encryption. The results are in figures 5.8 and 5.9. Once again, we conclude that the overhead does not grow with the problem size.

The most important observation from the measurements is that the performance impact of obfuscations does not depend on the problem size. Although the results vary, we think that all obfuscations add just a constant overhead to the program – meaning that all obfuscations are free in the scale in [1].

In the figures, we can also see a varying influence of obfuscations on program performance. We can notice that the combined performance impact of inlining, bogus control flow, interleaving and identifier stripping (without splitting BBs) adds a very little overhead to the program – the difference between obfuscated and non obfuscated program does not exceed factor 2 in all measurements we performed.

When outlining or splitting BBs is enabled, the results start to have a bigger variance. In the potency evaluation, we have seen that splitting BBs increases the potency of several obfuscations. Unfortunately, here we also see that it makes the performance impact more unpredictable. Outlining with basic splitting disabled causes a bigger variance too. The reason why just these two transformations cause this variance is that they are the only obfuscations that target a particular piece of code. The other obfuscations do not impact just a single piece of code, but rather process the whole function in the same manner.

We conclude that all obfuscations we implemented are free, the performance overhead they add does not grow with the problem size. Outlining and helper pass for splitting BBs create significant variance in different runs of the obfuscator. The reason is that sometimes they target a performance-critical code part, which results in bigger performance overhead. We tried to reduce that effect by decreasing obfuscations of inner loops; however, that is not perfect. We suggest that this could be solved by giving the user a better control over the obfuscation process (e.g., by selecting which code should be less obfuscated).

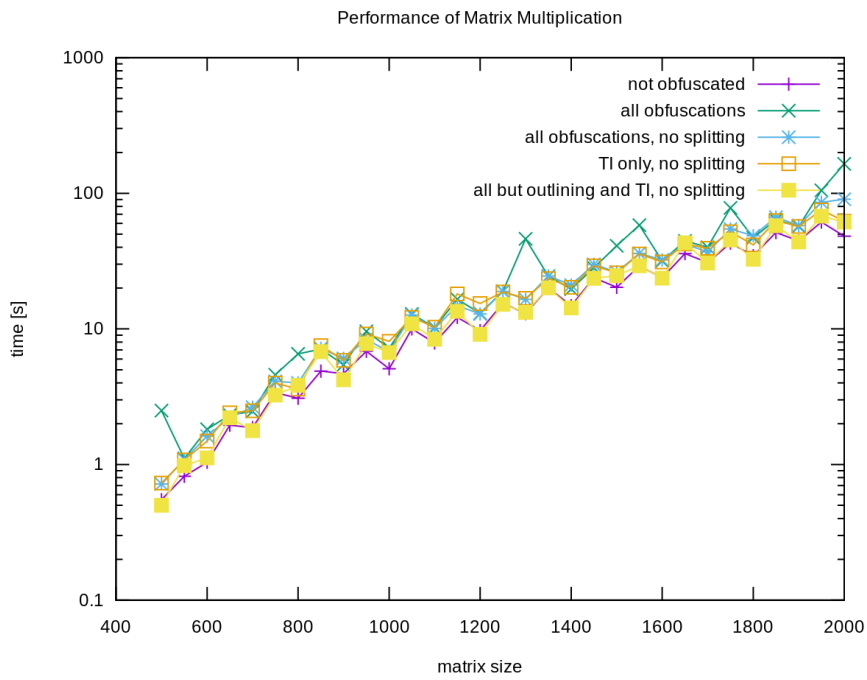


Figure 5.4: Matrix multiplication performance

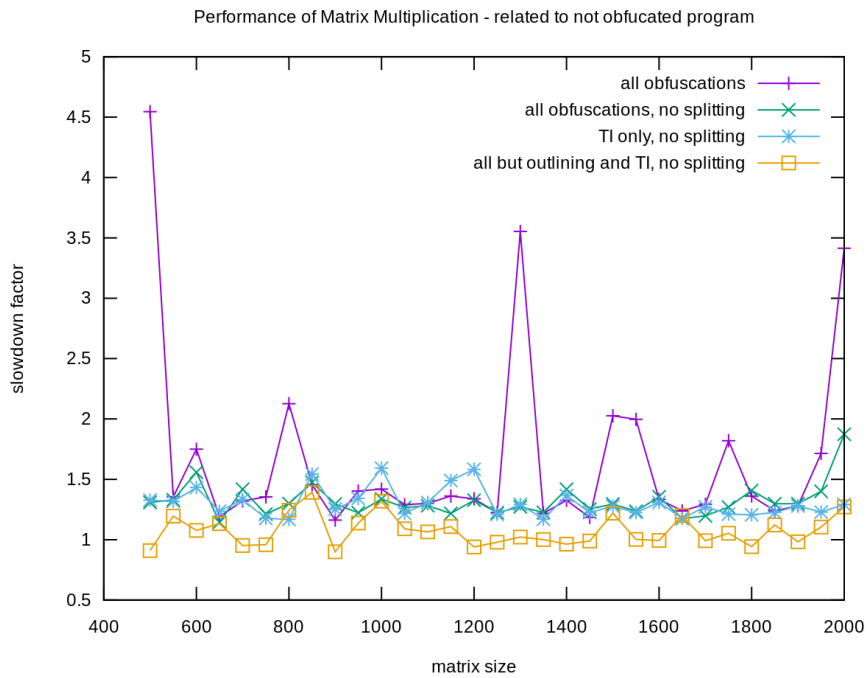


Figure 5.5: Matrix multiplication performance – related to non obfuscated program

## 5. EVALUATION

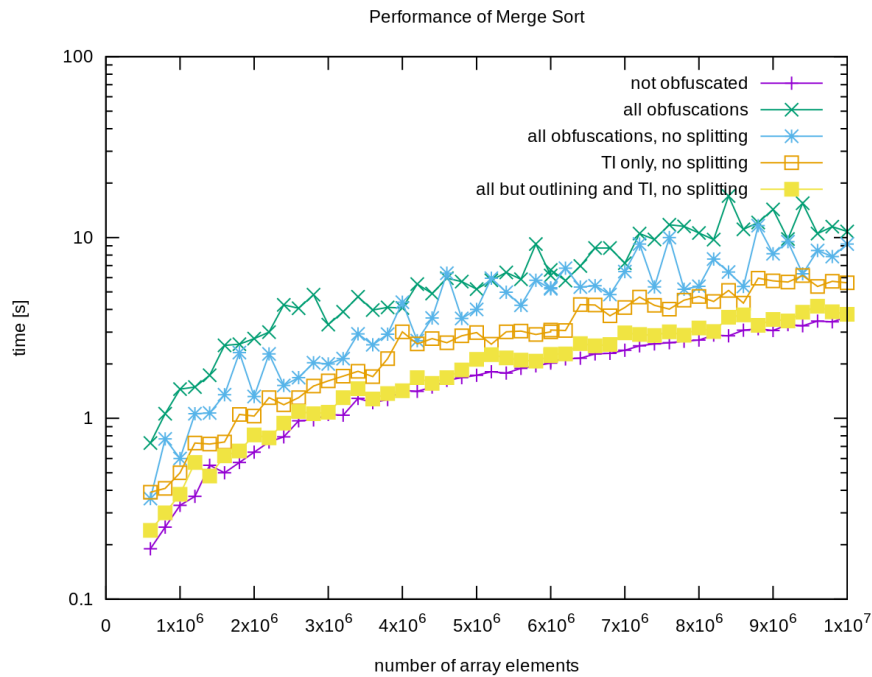


Figure 5.6: Merge sort performance

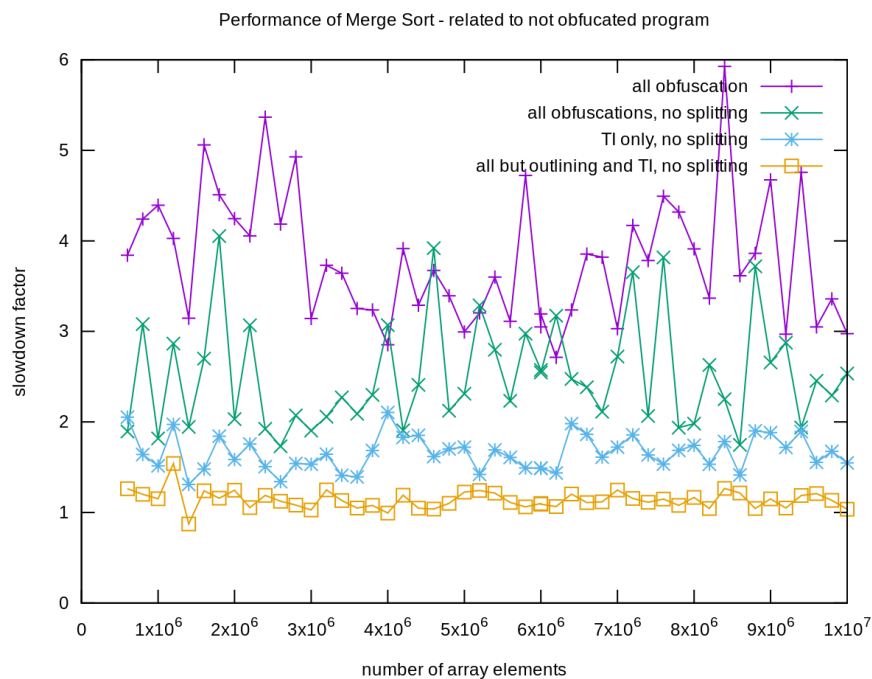


Figure 5.7: Merge sort performance – related to non obfuscated program

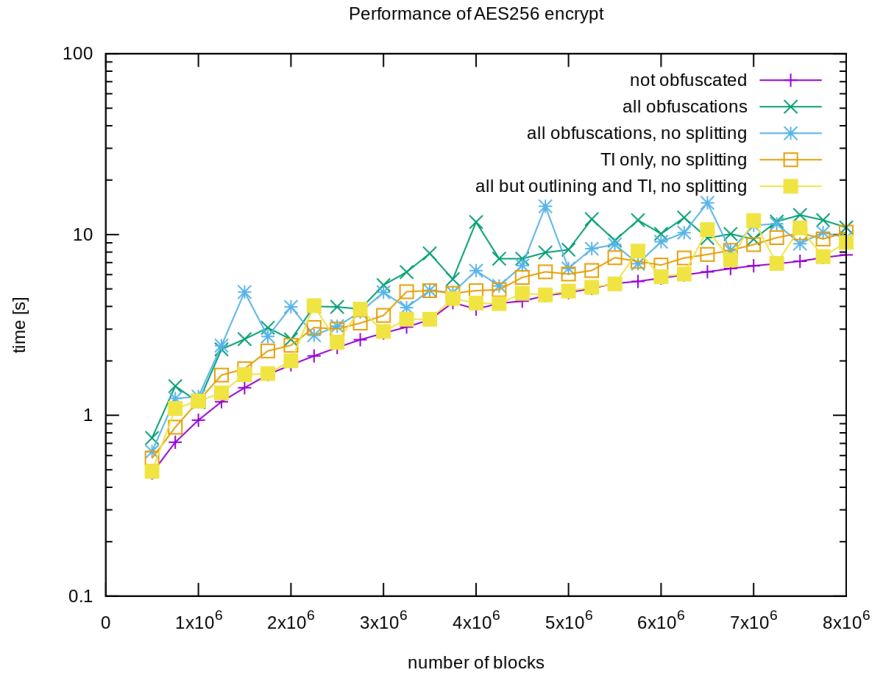


Figure 5.8: AES performance

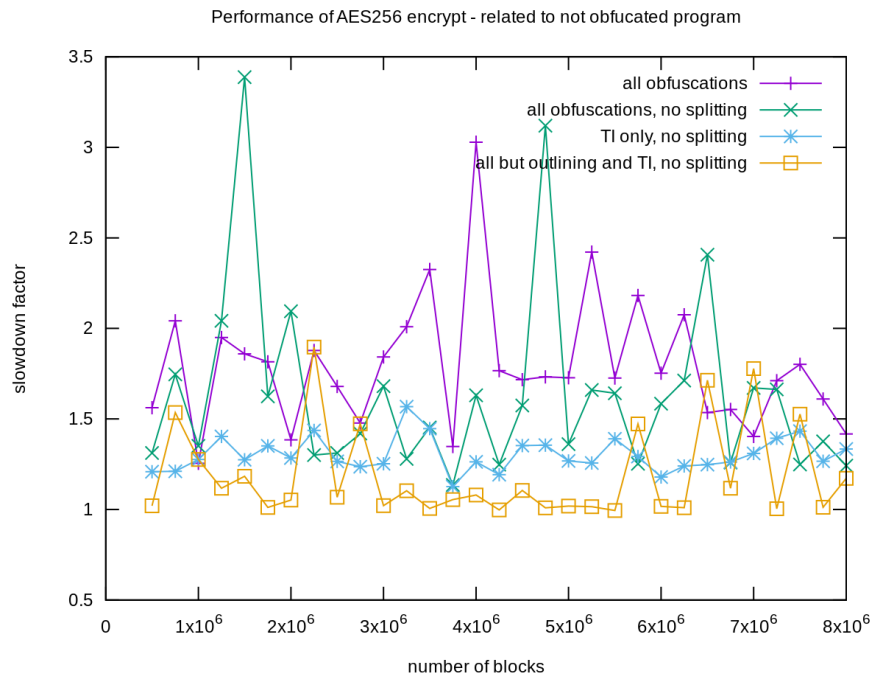


Figure 5.9: AES performance – related to non obfuscated program

### 5.1.4 Summary

In the previous sections, we have evaluated implemented obfuscations in terms of potency, resilience and performance impact. We assigned the following evaluation to implemented obfuscations:

Table 5.4: Evaluated obfuscation properties

<b>Obfuscation</b>	<b>Potency</b>	<b>Resilience</b>	<b>Cost</b>
Inlining	medium	one-way	free
Bogus Control Flow	low	weak	free
Outlining	low	strong	free
Function interleaving	low	strong	free
Table interpretation	medium	weak	free
Removing identifiers	medium	one-way	free

The results mostly correspond with the evaluation from [1], except of the table interpretation. Table interpretation is described as having high potency, strong resilience and being costly (adding a polynomial performance overhead to the program). We assume that they meant more complex implementation of that transformation. The obfuscation we have implemented is simpler and corresponds more to CFG flattening, described in [18].

## 5.2 Comparison with other obfuscators

In chapter 2 we reviewed several available obfuscators. In this chapter, we compare them to our obfuscator. We focus on comparison with Obfuscator-LLVM.

CXX-OBFUS works just on lexical level, so its obfuscation will not change the metrics. Its main feature is removing names – an obfuscation with medium potency and one-way resilience (according to [1]). Our obfuscator implements this obfuscation too and also implements several other transformations. We thus say that our obfuscator is more advanced than this tool. However, we see one advantage of this tool – it can scramble identifiers across modules. We are not able to do that in some cases, as described in 4.2.1.

StarForce C++ Obfuscator does not provide much detail about its functionality, and we were not able to test it. Its website, however, provides a code example. The example code seems to be processed by some control flow transformation – the control flow of the program has been completely changed. This obfuscation is none of those we have implemented. Thus, obfuscations performed by this tool might be more advanced than ours. We are, however, not able to test it better.

Tigress obfuscator offers more obfuscations that have both higher potency and resilience. It offers stronger opaque predicates as well as stronger obfusca-

tions, such as dynamically building functions at runtime (JIT). Our obfuscator is far behind that level. However, this tool is not intended for automatic obfuscation of the whole program. It has to be fine-tuned by the user. The user has to select which transformation should be used and where it should be applied. Also, this tool is limited just to C programming language.

### 5.2.1 Obfuscator-LLVM

We are able to perform the most detailed comparison with Obfuscator-LLVM, since it is open-source and it is based on the same principle. Obfuscator-LLVM offers three obfuscations: bogus control flow, instruction substitution and table interpretation (they call it CFG flattening). Let us look at the common obfuscations – bogus control flow and table interpretation.

Bogus control flow is implemented similarly as ours – it makes a copy of one basic block and alters it. An opaque predicate is used to guarantee that control flow always follows the desired way. It uses global variables and invariant expressions as well – but only one type of invariant expression ( $x(x+1)\%2 == 0$ ). The value of these global variables is not updated. Our obfuscator uses five different expressions and picks one of them randomly each time. It also randomly updates global variables from many locations, with an aim to make tracking their value difficult.

Obfuscator-LLVM implements table interpretation in its basic form. Our obfuscator implements several improvements to make it more potent and resilient – including the use of opaque predicates and adding invalid states to the jump table.

Our obfuscator also implements inlining, outlining, function interleaving and identifier removing. On the other hand, it does not implement instruction substitution, that is implemented in Obfuscator-LLVM. Nevertheless, the potency of instruction substitution obfuscation is low.

Overall, we think our implementation offers more advanced features than Obfuscator-LLVM. Its strongest obfuscation is table interpretation, with medium potency and strong resilience. Our obfuscator features several obfuscations having medium potency and some transformations that have one-way resilience.

On the other hand, we should note that Obfuscator-LLVM is likely better tested (as it is available for several years) and allows the user to control the obfuscation process better – the user can annotate functions that should not be obfuscated.





---

# Conclusion

The aim of this work was to i) design and implement an automatic obfuscator based on LLVM compiler infrastructure, ii) evaluate potency and resilience of implemented obfuscation transformations and iii) compare results with other similar obfuscations tools.

We have explored and described several obfuscation transformations and metrics to evaluate them. We have researched similar LLVM-based obfuscation tool. Then, we have designed our own obfuscation transformations and their integration into LLVM allowing obfuscations to be language-independent.

Along with obfuscation transformations, we performed analysis of the designed transformations and standard obfuscation metrics, i.e., potency, resilience, and performance impact of implemented obfuscations on code being obfuscated. We have evaluated our obfuscator, and our results suggest that: i) our obfuscator manages to make the program more confusing for a potential reverse engineer, ii) removing these obfuscations automatically would require non-trivial work, and iii) all obfuscations only add a constant performance overhead to the program. We have furthermore demonstrated that our obfuscator is language-agnostic by testing it on programs in different programming languages such as C, C++, and Rust.

Our obfuscator was compared to other available similar obfuscation tools. The comparison has indicated that our obfuscator was more advanced than the tools mentioned.

## Possible future work

We see several possible ways to improve or extend our obfuscator. Obfuscations can be made more potent. For example, table interpretation can be made more potent by making the selection of the next basic block more confusing – articles [18, 16] suggest ways how this can be achieved using global pointers or one-way functions. Resilience can be improved by using stronger opaque predicates – for example, article [19, 5] suggest such ways. In section 5.1.3 we

## CONCLUSION

---

have shown that the performance impact varies and the performance penalty is higher when obfuscations target a performance-critical code part. This performance impact unpredictability would be undesirable for practical use of our obfuscator. We suppose that it could be reduced by giving the user a fine-grained control over the obfuscation process, e.g., by marking code parts that should be less obfuscated by means of language pragma or attribute statements.

## Acronyms

**API** Application Programming Interface

**BB** Basic Block

**CFG** Control Flow Graph

**LLVM** Low Level Virtual Machine



---

# Configuration options

Our obfuscator offers various configuration options. These options have their default value, that can be changed by the user. The user can override default values by setting an environment variable.

## B.1 Scheduler pass

Scheduler pass is used to run other obfuscations in order described in 4.1.6. Configuration options allow to disable some obfuscations:

- `OBF_DISABLE`: disables all obfuscations
- `OBF_DISABLE_INLINING`: disables inlining obfuscation
- `OBF_DISABLE_SPLIT`: disables splitting blocks
- `OBF_DISABLE_BCF`: disables bogus control flow obfuscation
- `OBF_DISABLE_OUTLINING`: disables outlining obfuscation
- `OBF_DISABLE_INTERLEAVING`: disables function interleaving
- `OBF_DISABLE_TABLE`: disables table interpretation obfuscation
- `OBF_DISABLE_OPAQUE`: disables opaque predicates pass
- `OBF_DISABLE_STRIP`: disables removing identifiers

## B.2 Obfuscation passes

Most of the obfuscation passes offer some configuration options. There options are listed here:

## B. CONFIGURATION OPTIONS

---

- `OBF_INLINE_LIMIT`: size limit for obfuscation. A multiple of the original program size. Inlining will stop when the program size reaches that limit. The default value is 4.
- `OBF_SPLIT_FACTOR`: split factor, sets the target number of basic blocks. The default value is 2 (i.e., the resulting number of basic blocks will be  $factor * originalNum$ )
- `OBF_BOGUS_FLOW_PROB`: probability of inserting bogus flow. Each found unconditional branch instruction will be turned into conditional with this probability. The default value is 0.25.
- `OBF_OUTLINE_MAX_FNS`: maximum number of functions to outline (from each function). Default value 10.
- `OBF_OUTLINE_MIN_SIZE`: minimum size of function after outlining. Specified as a fraction of original function size. The default value is 0.5 (i.e., a function may be reduced to half of its original size).
- `OBF_OUTLINE_ATTEMPTS`: number of attempts to select a suitable region for outlining. Inlining is stopped if no suitable region is found. The default value is 200.
- `OBF_INTERLEAVE_PASSES`: number of interleaving passes. Each function can be interleaved at most once in each pass, so this also limits how many functionalities can be contained within one function. The default value is 3.
- `OBF_INTERLEAVE_OPAQUE_PROB`: probability that the call to interleaved function will be protected by opaque predicate. The default value is 0.2.
- `OBF_INTERLEAVE_MAX_FNS`: maximum number of functions that can be interleaved in each pass. The default value is 10.
- `OBF_TABLEINTER_OPAQUE_PROB`: probability that a jump to dispatcher will be protected by opaque predicate, to make tracking values more difficult. The default value is 0.3.
- `OBF_OPAQUE_VAR_NUM`: number of global variables to use for opaque predicates. One of them is randomly chosen each time when invariant expression is evaluated. The default value is 8.

---

## Bibliography


- [1] Collberg, C.; Thomborson, C.; et al. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
- [2] Allen, F. E. Control flow analysis. In *ACM Sigplan Notices*, volume 5, ACM, 1970, pp. 1–19.
- [3] Yi, D. A New Obfuscation Scheme in Constructing Fuzzy Predicates. In *Software Engineering, 2009. WCSE'09. WRI World Congress on*, volume 4, IEEE, 2009, pp. 379–382.
- [4] Xu, D.; Ming, J.; et al. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *International Conference on Information Security*, Springer, 2016, pp. 323–342.
- [5] Majumdar, A.; Thomborson, C. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, Australian Computer Society, Inc., 2006, pp. 187–196.
- [6] Ming, J.; Xu, D.; et al. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 757–768.
- [7] Prakash, R. K. R.; Amritha, P.; et al. Opaque Predicate Detection by Static Analysis of Binary Executables. In *International Symposium on Security in Computing and Communication*, Springer, 2017, pp. 250–258.
- [8] Drape, S. Intellectual property protection using obfuscation. 2010.
- [9] Majumdar, A.; Thomborson, C.; et al. A survey of control-flow obfuscations. In *International Conference on Information Systems Security*, Springer, 2006, pp. 353–356.

- [10] Chow, S.; Gu, Y.; et al. An approach to the obfuscation of control-flow of sequential computer programs. In *International Conference on Information Security*, Springer, 2001, pp. 144–155.
- [11] László, T.; Kiss, Á. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, volume 30, 2009: pp. 3–19.
- [12] Banescu, S.; Collberg, C. S.; et al. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 189–200. Available from: <http://dl.acm.org/citation.cfm?id=2991114>
- [13] Junod, P.; Rinaldini, J.; et al. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015*, edited by B. Wyseur, IEEE, 2015, pp. 3–9, doi:10.1109/SPRO.2015.10.
- [14] Lattner, C. The architecture of open source applications: LLVM. 2014.
- [15] Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 75.
- [16] Cappaert, J.; Preneel, B. A general model for hiding control flow. In *Proceedings of the tenth annual ACM workshop on Digital rights management*, ACM, 2010, pp. 35–42.
- [17] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, , no. 4, 1976: pp. 308–320.
- [18] Wang, C.; Davidson, J.; et al. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, IEEE, 2001, pp. 193–202.
- [19] Collberg, C.; Thomborson, C.; et al. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1998, pp. 184–196.




---

## Contents of enclosed medium

 DP\_Petracek\_Martin\_2018

 README.txt — the file with content description


 SRC — the directory of source codes


 OBFUSCATOR — implementation sources

 THESIS — the directory of L<sup>A</sup>T<sub>E</sub>X source codes of the thesis

 TESTS — the directory of test programs

 TEXT

 thesis.pdf

 thesis.ps