



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: LLVM Obfuscator Based on Virtual Machines with Custom Opcodes and String Encryption
Student: Bc. Lukáš Turčan
Supervisor: Ing. Tomáš Zahradnický, Ph.D.
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Information Security
Validity: Until the end of winter semester 2019/20

Instructions

Explore the taxonomy of obfuscation transformations [1]. Get acquainted with the LLVM framework [2] and its internal representation (IR) format. Design and implement a set of obfuscation transformations that will do lightweight virtualization, encrypt strings, shuffle basic blocks, and interleave an unlimited number of functions to make the program more resilient against reverse engineering. The implemented LLVM module must support the `#pragma` directive to control obfuscation parameters from the source code when necessary. Analyze potency and resilience of each implemented transformation. Compare the results with other available obfuscation tools. In the end, bundle the created transformations together with selected transformations from theses [3] and [4] into a single tool.

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/>.
- [3] Petráček, Martin. LLVM Obfuscator. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.
- [4] Šíma, Roman. Obfuscátor nad platformou LLVM. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

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

Prague September 27, 2018



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

LLVM Obfuscator Based on Virtual Machines with Custom Opcodes and String Encryption

Bc. Lukáš Turčan

Department of Information Security

Supervisor: Ing. Tomáš Zahradnický, EUR ING, Ph.D.

May 9, 2019

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Zahradnický, EUR ING, Ph.D., for his helpful advice and overwhelming patience for the time he was helping me with this thesis. Also, I am grateful for the wonderful people on the LLVM Slack Channel who were helping me all the time to overcome my struggles with LLVM.

Finally, I must express my gratitude towards my family and friends who were supporting me throughout my studies.

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 for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 9, 2019

.....

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Lukáš Turčan. 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

Turčan, Lukáš. *LLVM Obfuscator Based on Virtual Machines with Custom Opcodes and String Encryption*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Obfuskace je jednou z metod ochrany duševního vlastnictví proti pirátství, kopírování nebo nechtěné modifikaci. Cílem této práce je vytvořit automatický obfuskátor založený na Frameworku LLVM, jehož modularita umožňuje implementaci obfuskačních transformací nezávislých na zdrojovém jazyce a cílové architektuře. Hlavním úspěchem této práce je důkaz, že je možné použít Framework LLVM k vytvoření transformací založených na virtualizaci. Implementované transformace vedle virtualizace zahrnují šifrování řetězců a vylepšené prokládání funkcí.

Klíčová slova obfuskační transformace, reverzní inženýrství, LLVM, virtuální stroje, opkódy, šifrování řetězců

Abstract

Obfuscation is one of the methods for protecting intellectual properties against piracy, copying or tampering. This thesis aims to create an automatic obfuscator based on the LLVM Framework, whose modularity allows implementing obfuscation transformations agnostic to the source language and target architecture. A major achievement of this thesis is the proof that it is possible to use the LLVM Framework to create virtualization-based transformations. Implemented transformations also include string encryption and improved function interleaving.

Keywords obfuscation transformations, reverse engineering, LLVM, virtual machines, opcodes, string encryption

Contents

Introduction	1
1 Obfuscation Transformations	3
1.1 Definition	3
1.2 Basic Terms	4
1.3 Evaluation of Obfuscating Transformations	6
1.4 Outline of Obfuscation Techniques	9
1.5 Computations Transformations	9
1.6 Aggregation Transformations	12
1.7 Ordering Transformations	13
1.8 Data Transformation	13
1.9 Summary	14
2 Existing Tools	15
2.1 Commercial Tools	15
2.2 Open-source Tools	17
2.3 Summary	19
3 Design	21
3.1 LLVM Framework	21
3.2 Transformation Passes	28
3.3 Grain Control of Obfuscation Passes	31
3.4 Summary	32
4 Implementation	33
4.1 Lightweight VM Pass	33
4.2 String Encryption Pass	36
4.3 Improved Interleaving Pass	39
4.4 Summary	40

5	Evaluation	41
5.1	Obfuscation Metrics	41
5.2	Evaluation of String Encryption Pass	49
5.3	Quality of Lightweight VM	52
5.4	Comparison with Existing Tools	52
5.5	Summary	54
6	Conclusion	55
6.1	Future Work	56
	Bibliography	57
A	Acronyms	59
B	Usage	61
B.1	Getting Started	61
B.2	Available Passes & Parameters	62
B.3	String Encryption Pass	65
B.4	Debug Mode	65
C	Contents of enclosed CD	67

List of Figures

1.1	Example of Control Flow Graph, instructions were omitted.	4
1.2	An example of opaque constructs.	5
1.3	Table Interpretation Example, (a) is the original CFG and (b) is the CFG after transformation.	10
1.4	Process of VM obfuscation. (1) Firstly, we disassemble the code into native assembly code (1). (2) The assembly code is mapped into a custom instruction set. (3) Then is encoded into a bytecode format. (4) Finally, the generated bytecode is inserted into the binary. [7]	11
3.1	Classical Compiler Design [8]	21
3.2	Design of a compiler with multiple front-ends and back-ends [8] . .	22
3.3	LLVM's implementation of Three-Phase Design [8]	22
3.4	C function represented in LLVM IR. Note: some instructions were wrapped for a better readability.	23
3.5	A comparison of two approaches for obtaining value from preceding BBs. Note: code was optimized for better readability.	24
3.6	An example of assigning opcodes.	28
3.7	An example of pragmas. Note: pragmas are one-liners, the code was wrapped for better readability.	32
4.1	An example of strings used in multiple arrays. Note: the code was optimized for better readability.	38
5.1	AVL Tree Elements Insertion performance	46
5.2	AES Encrytion performance	47
5.3	QuickSort performance	48
5.4	Decryption performance — Printing Speed	50
5.5	Decryption performance — Printing Slowdown	51

5.6	Comparison of binary sizes. The ratio of binary sizes with encrypted strings to binary sizes with plain strings.	51
5.7	Visualization of AVL's insert function in IDA. Non-obfuscated version.	52
5.8	Visualization of AVL's insert function in IDA. VM-obfuscated with default options.	53
5.9	Visualization of AVL's insert function in IDA. VM-obfuscated with custom switch and non-succeeding opcodes.	53

List of Tables

1.1	Overview of some popular software complexity measures. $E(X)$ is the complexity of a software component X . F is a function or method, C a class, and P a program. [2]	7
1.2	Definition of resilience. [2]	8
5.1	Program length changes (μ_1) for selected programs. Total is a sum for all functions, and Max is the maximum value among functions. Value is a ratio of obfuscated and non-obfuscated program's metrics. NSO stands for non-succeeding opcodes and unlimited stands for merge unlimited functions together. See B.2 for parameter explanation.	42
5.2	Cyclomatic complexity changes (μ_2) for selected programs. Total is a sum for all functions, and Max is the maximum value among functions. Value is a ratio of obfuscated and non-obfuscated program's metrics. NSO stands for non-succeeding opcodes and unlimited stands for merge unlimited functions together. See B.2 for parameter explanation.	43
5.3	Evaluated obfuscation metrics. Performance impact of String Encryption was measured in section 5.2	49

Introduction

Naturally, software creators strive for methods to protect their intellectual properties against piracy, tampering or copying. The motivation emerges from investments into expensive development of their products. Even further, they may invent a new game-changing algorithm or another kind of asset that requires a higher level of protection. For instance, most sales in the video gaming industry occur during the first weeks after the release. Thus it is crucial to protect their product as much as possible to delay the development of modification (crack) that enables illegal use without a license. Piracy groups responsible for developing cracks are even challenging themselves to be the first in cracking a particular application.

One of the protection methods is keeping a crucial part of the software on a server-side and only providing access to in the form of a thin client-side application. This approach is often impractical for various reasons. For instance, it requires a permanent Internet connection and consumes bandwidth. Another issue is that this method is currently unusable for applications requiring to transfer large amounts of data, for example, the size of Adobe Photoshop is more than one gigabyte, and the distribution of all its features from the server-side would be too slow, possibly leading to lags even with the high-speed connections nowadays widely available. In some cases, software vendors choose a trade-off, where an application has a part of its features bundled in the client-side, and the rest is provided through the server-side.

Due to mentioned discomforts and technical limitations, many software companies and users prefer fully-featured client applications. The main dividing line for protection methods of client applications is hardware and software. Hardware protection is commonly in the form of a special device (also referred to as dongle or hardware key) programmed with a product-key or cryptographic protection mechanism that has to be plugged in to run software properly. Even though this method is conventional for expensive programs, distributing hardware-keys would be impractical for massively-distributed software. In such a case, companies have to rely on software protection methods.

The available software protections are obfuscations, watermarking and tamper-proofing. Obfuscation modifies a program in a way that makes it more resilient against reverse-engineering while keeping its original functionality. Watermarking is a process of unique-copy distribution of an application with a user identifier to trace the origin of its piracy. Tamper-proofing protects software against unauthorized modifications (e.g., the removal of a watermark), by causing them to result in malfunctioned software.

This thesis deals with obfuscation transformations. Barak et al. [1] proved the impossibility of creating an unbreakable obfuscation and with sufficient competency, time and energy given, every software can be reverse-engineered. We see the goal of obfuscations as entirely different. By making software hard to reverse-engineer, we try to discourage a potential reverse engineer from investing his time into such feat. We aim to make reverse engineering economically unfeasible.

The possible applications of obfuscations also include nefarious ones, such as malware obfuscation. Malware is often obfuscated either to make analyses conducted by antivirus companies as difficult as possible or because it exploits some unknown vulnerability (0-day) and is beneficial to the author to delay the discovery of the vulnerability by security researchers. Nevertheless, there are still many valid reasons to perform research in this area. The knowledge of obfuscation techniques is inevitable for a reverse-engineer to be able to deobfuscate malware, and there are many legitimate uses of obfuscations mentioned at the beginning of this introduction.

Manual obfuscations are time-consuming and complicate deployment process. Therefore, it is more convenient when the obfuscations are done automatically as a part of the deployment process. Thus, we decided to implement an automatic obfuscator based on the LLVM Framework. Nowadays, the LLVM Framework is very popular and leveraged by many tools such as Xcode to perform optimizations on Swift code. LLVM supports a variety of languages (C, C++, Obj-C, Swift, etc.) and its modularity allows us to create transformations that are agnostic to a source language and target architecture.

The structure of this thesis is organized as follows: the first chapter explains theoretical basics and introduces obfuscation transformations, the second chapter summarizes existing obfuscation tools, the third chapter presents the LLVM Framework and puts forward the design of our obfuscator, the fourth chapter deals with the details of our implementation, the fifth chapter evaluates the created tool and the last chapter summarizes this thesis.

Obfuscation Transformations

In this chapter, we formally introduce obfuscation transformations and evaluation metrics. Then we explain basic concepts and several obfuscation techniques. Informally, obfuscation transformations make the software more resilient against reverse-engineering but preserve its original functionality.

1.1 Definition

Collberg et al. [2] defined obfuscation in the following way:

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 .*

Observable behavior is defined loosely as “behavior as experienced by the user”. This means that P may have side-effects (such as creating files, sending messages over the Internet, etc) that P does not, as long as these side effects are not experienced by the user. Note that we do not require P and P' to be equally efficient.

Collberg et al. [2] set the main dividing line of obfuscation transformations to be the kind of information it targets:

- **Layout obfuscations** remove the source code formatting, scramble identifier names and remove formatting.

- **Data obfuscations** aggregate variables/arrays, change orders of arrays/variables/methods, encrypt data, convert static data to procedures.
- **Control obfuscations** change program's flow, for example by adding conditional branches based on opaque predicates with unreachable code, inlining to or outlining from functions.
- **Preventive obfuscations** are aimed against automatic tools. They try to make automatic deobfuscation techniques more difficult or exploit known issues in deobfuscators or decompilers.

1.2 Basic Terms

1.2.1 Function Representation

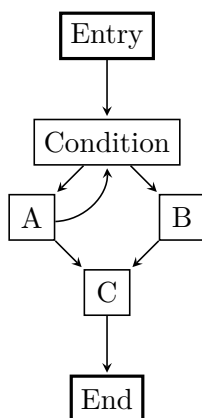


Figure 1.1: Example of Control Flow Graph, instructions were omitted.

A function is set of Basic blocks [4, 10], where Basic Block (BB) is a sequence of linear instructions, with one entry point (the first executed instruction) and one exit point (the last executed instruction — terminator instruction). The exit point, also called terminator instruction, is either a jump to another BB or a return from a function. If BB A jumps to BB B, we say that A is the predecessor of B and B is the successor of A. The first executed BB in a function is called entry BB, and last one(s) called terminating BB. Each function has exactly one entry BB and at least one terminating BB.

A function can be represented as a directed graph called Control Flow Graph (CFG), where nodes are BBs and edges denotes predecessors and successors.

In Figure 1.1 we can see an example of a CFG. *Condition* has one predecessor *Entry* and two successors, *A* and *B*. *Condition* is also a predecessor

of A . C has two predecessors A , B and one successor End . Finally, End is a terminating BB and it returns from the function.

1.2.2 Opaque Constructs

The quality of transformations that obfuscate control flow directly depends on opaque constructs. The goal of opaque constructs is to make transformations resistant to attacks from deobfuscators. To achieve this we use *opaque variables* and *opaque predicates*.

- **An opaque variable** has some property known to the obfuscator but difficult for deobfuscator to deduce.
- **An opaque predicate** has its outcome known to the obfuscator but difficult for deobfuscator to deduce.

1.2.2.1 Trivial and Weak Opaque Constructs

<pre>int x = 5, y = 3; if(x > 2){ // Always executed. } if(random(2,5) < 0){ // Never executed. }</pre>	<pre>int x = 5, y = 3; if(y < 7){ // Always executed. --x; } if(x > 4){ // Never executed. }</pre>
--	---

(a) trivial

(b) weak

Figure 1.2: An example of opaque constructs.

Collberg et al. [2] defined trivial and weak opaque constructs as:

- **Trivial Opaque Construct** can be cracked by deobfuscator using static local analysis. Static local analysis is an analysis within a single BB.
- **Weak Opaque Construct** can be cracked by deobfuscator using static global analysis. Static global analysis is analysis within a single CFG.

In Figure 1.2 we can see a comparison of weak and trivial opaque constructs. In (a) we can crack opaque constructs just by looking at standalone conditions because variables in conditions are not changed in previous BBs. On the contrary, to analyze the second condition in (b), we need to evaluate the first condition to know if the variable x was changed and consequently we can evaluate the second one.

1.2.2.2 Aliasing Predicates

Predicates to be useful should be created in polynomial time and should require exponential time to be cracked (according to the size of the program). Collberg [3] proposed that this can be achieved by aliasing.

Aliasing concept means creating a set of complex data structures (heaps, trees, graphs, etc.) and keeping several pointers into them. Then the created structures are occasionally updated (adding/removing elements, splitting/merging structures and so on), while maintaining invariants. Such invariants are for instance: $p1$ points to a heap with a maximum element equal to 100 , $p2$ refers to a node that is never a predecessor of $p3$, there is always a path from $p4$ to $p5$, and so on.

Static analyses of aliasing constructs were proved to be NP-hard [6], and we can easily create invariants that can be tested in a constant time, which gives us cheap bogus code.

1.3 Evaluation of Obfuscating Transformations

Before we introduce several transformations, it is desirable to be able to evaluate their quality. In this section, we will introduce metrics defined in [2].

1.3.1 Potency

Definition 2. Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P' . Let $E(P)$ be the complexity of P , as defined by one of the metrics in Table 1.1.

$\tau_{pot}(P)$, the potency of τ with respect to a program P , is a measure of the extent to which τ changes the complexity of P . It is defined as:

$$\tau_{pot}(P) = \frac{E(P')}{E(P)} - 1.$$

τ is a potent obfuscating transformation if $\tau_{pot}(P) > 0$.

Informally, we can say that potency describes how much additional work of a human is required to understand an obfuscated program than its non-obfuscated equivalent. Potency is measured on a three-point scale: **low**, **medium** and **high**.

Metric	Metric Name	Description
μ_1	Program Length	$E(P)$ increases with the number of operators and operands in P .
μ_2	Cyclomatic Complexity	$E(F)$ increases with the number of predicates in F .
μ_3	Nesting Complexity	$E(F)$ increases with the nesting level of conditionals in F .
μ_4	Data Flow Complexity	$E(F)$ increases with the number of inter-basic block variable references in F .
μ_5	Fan-in/out Complexity	$E(F)$ increases with the number of formal parameters to F , and with the number of global data structures read or updated by F .
μ_6	Data Structure Complexity	$E(P)$ increases with the complexity of the static data structures declared in F . The complexity of an array increases with the number of dimensions and with the complexity of the element type. The complexity of a record increases with the number and complexity of its fields.

Table 1.1: Overview of some popular software complexity measures. $E(X)$ is the complexity of a software component X . F is a function or method, C a class, and P a program. [2]

1.3.2 Resilience

From the definition of **potency**, it would seem that we can effortlessly increase μ_1 or μ_2 metrics by adding dead code. Unfortunately, these transformations are useless as there are existing transformations that can easily undo this operation.

For this reason, another metric must be introduced. This metric is called **resilience** and means how much programmer’s effort we need to create an automatic deobfuscator and how much computation resources the obfuscator will require to undo a transformation.

It is difficult to estimate a programmer’s effort, and we have to come up with something that is easily to evaluate, even it is not always 100% accurate. In [2] programmer’s effort is measured on a four-point scale:

- **Local:** if it affects a single basic block

	Deobfuscator's effort	
Programmer's effort	Polynomial	Exponential
Local	trivial	weak
Global	weak	strong
Inter-procedural	strong	full
Inter-process	full	full

Table 1.2: Definition of resilience. [2]

- **Global:** if it affects an entire function
- **Inter-procedural:** if it affects the flow of information between functions
- **Inter-process:** if it affects the interaction between independently executed threads

Definition 3. Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P' . $\tau_{res}(P)$ is the resilience of τ with respect to a program P .

$\tau_{res}(P)$ is one-way if information is removed from P and cannot be reconstructed from P' . Otherwise, $\tau_{res}(P)$ is defined in table 1.2.

Resilience is measured on a five-point scale: **trivial**, **weak**, **strong**, **full**, **one-way**.

1.3.3 Resource Impact

The obfuscated program needs more resources (CPU time, space, memory) than its non-obfuscated variant. The obfuscation transformation is a trade-off between the number of resources and quality of the obfuscation. In [2] the impact is measured on the following scale:

- **Dear:** if executing the obfuscated variant requires exponentially more resources than regular one,
- **Costly:** if executing the obfuscated variant requires $\mathcal{O}(n^p)$, $p > 1$ more resources than regular one,
- **Cheap:** if executing the obfuscated variant requires $\mathcal{O}(n)$ more resources than regular one,
- **Free:** if executing the obfuscated variant requires $\mathcal{O}(1)$ more resources than the regular one.

1.4 Outline of Obfuscation Techniques

In [2] are obfuscations divided as follows.

Layout Transformations are trivial transformations such as removing comments, source-code formatting and scrambling identifier names.

Control Transformations alters control-flow and are divided as:

- **Computations Transformations** insert dead or duplicate code in order to hide the real control-flow, remove control-flow abstractions or introduce fake abstractions.
- **Aggregation Transformations** include inlining, outlining, interleaving or cloning methods.
- **Ordering Transformations** are explained as follows. Programmers tend to organize their source code in a way that logically related things are physically close. Ordering Transformations try to break-up these relationships by changing the order in which operations are performed.

Data Transformations obscure data and data structures that a program may use.

1.5 Computations Transformations

1.5.1 Insert Dead or Irrelevant Code

There is a strong correlation between the perceived complexity of code and the number of predicates it contains. Some examples of irrelevant code insertion:

- An opaque predicate p^T that always evaluate to true. False branch will lead to dead code.
- An opaque predicate $p^?$ that sometimes evaluate to true and sometimes to false. The true and false branches will contain the same but differently obfuscated code.
- An opaque predicate p^T that always evaluate to true. The true will contain the correct code and false branch same code with some bugs.

Potency and resilience were evaluated in [2] as depending on used opaque predicates.

1.5.2 Extend Loop Conditions

Loop's terminating condition can be obfuscated by adding an opaque predicate in a way that it will not change the number of cycles. For instance, we can add a predicate that always evaluates to true. Collberg et al. [2] also evaluated the quality of this transformation as depending on used opaque predicates.

1.5.3 Remove Standard Library Calls

Most programs rely on calls to standard libraries, and it can provide significant clues to a reverse engineer. We can replace them with our implementation with the same behavior. Unfortunately, this will increase the binary size. The transformation is estimated in [2] as having *medium* potency and *strong* resilience.

1.5.4 Table Interpretation

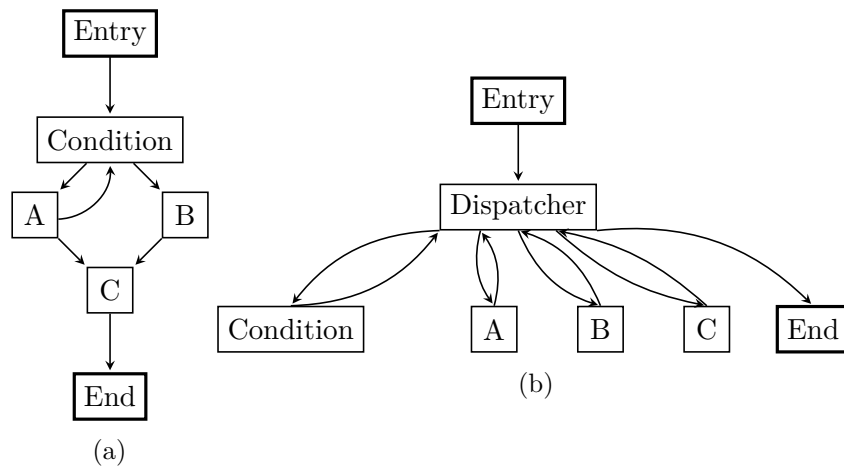


Figure 1.3: Table Interpretation Example, (a) is the original CFG and (b) is the CFG after transformation.

The complexity of a control flow determination is linear to the number of BBs [13]. Table Interpretation, also referred to Control Flow Flattening [11] or Dynamic Dispatcher [12], makes the determination harder by flattening BBs. Each BB is assigned an identifier and does not jump directly to its successor anymore but sets successor's identifier and jumps to a new BB called dispatcher. The dispatcher reads a value stored in a shared variable and then jumps to the selected BB according to the value. Table Interpretation brings some additional computing cost, created by jumps to dispatcher and dispatcher's workload.

In Figure 1.3 we can see an example of Table Interpretation transformation. Entry BB's successor was changed from *Condition* to *Dispatcher* and the rest BBs are flattened. We are not able to determine the real control flow in (b) without analyzing instructions.

1.5.5 Bogus Control Flow

Bogus Control Flow is a replacement of an unconditional jump to a BB with a conditional one based on an opaque construct. A new BB is either created by cloning existing BBs or filled-in with junk instructions, and the condition gives a misleading feeling that it may jump into the new BB that is never executed.

1.5.6 Virtual Machine Based Obfuscation

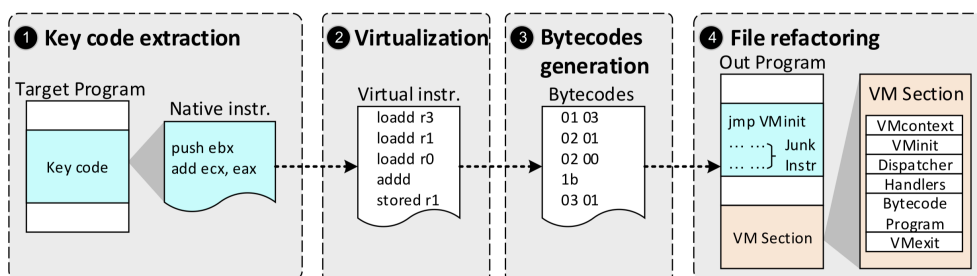


Figure 1.4: Process of VM obfuscation. (1) Firstly, we disassemble the code into native assembly code (1). (2) The assembly code is mapped into a custom instruction set. (3) Then is encoded into a bytecode format. (4) Finally, the generated bytecode is inserted into the binary. [7]

The idea of Virtual Machine (VM) Obfuscation is to convert a group of instructions into a custom instruction set unknown to attackers [7]. The instructions are then executed by an interpreter of the custom instruction set bundled with the application. In Figure 1.4, we can see a detailed process of VM obfuscation.

This is the extremely efficient protection, as a potential attacker has to understand the instruction set and create a custom disassembler. It is even possible to stack custom VMs, in this case, an interpreter does not translate directly the custom instruction set into native machine code but into another custom instruction set that has to be interpreted into native machine code.

This technique leads to a huge performance penalty, and we should use it only for small parts or parts that require a high level of protection. Collberg et al. [2] described virtualization as having *high* potency, *strong* resilience and *costly* obfuscation cost.

1.5.7 Redundant Operands

Another example of obfuscation transformation is adding redundant operands to arithmetic expressions. This technique is the most appropriate for integer values where we do not need to deal with numerical stability. See examples below.

- $X = X + V \xrightarrow{\tau} X = X + V * P$, where $P = 1$
- $Z = L + 1 \xrightarrow{\tau} Z = L + \frac{P}{2}$, where $P = 2Q$ and $Q = \frac{P}{2}$

The quality of redundant operands is evaluated in [2] as depending on the quality of used opaque predicate and nesting level.

1.6 Aggregation Transformations

Programmers use abstractions to overcome the complexity of software development. This allows us to obscure function calls. There are several ways in which methods can be obscured such as inlining, outlining, interleaving, and cloning.

1.6.1 Inline and Outline Methods

Function Inlining is the replacement of a call to function with the function's body. Function outlining is exactly the opposite when we move a piece of instructions into a new function, and instead of them, we put a call to this newly created function. Both methods do not bring much performance penalty but increase the binary size, and therefore, they should be used cleverly. They are very good one-way methods, as they destroy programmer's abstractions or even create opaque abstractions to confuse a reverse-engineer.

In [2] is inlining estimated as having *medium* potency, *one-way* resilience and *free* cost. Outlining is evaluated as having *medium* potency, *strong* resilience and *free* cost.

1.6.2 Interleave Methods

Function Interleaving is merging two or more functions into one, by merging their bodies and arguments with an extra parameter to select a correct behavior of the former function. Original functions are removed, and calls to them are replaced with calls to a new interleaved function.

We can add more confusion by selecting functions with similar arguments and reusing them in different contexts. It is also possible to add dead code by selecting functions that do not change the number of arguments of the interleaved function and copy their bodies without replacing calls. We will refer such functions as *dead functions*.

In [2] is the quality of interleaving evaluated as depending on used opaque predicates in the selection logic.

1.6.3 Clone Methods

Reverse engineers try to understand the purpose of a function by looking at places where it is being called. When the function is called at several places, then creating several copies of a function and calling function's copies instead of the original function makes this process more difficult. Function Cloning has no performance impact but leads to an increase of binary size.

Quality of cloning is described in [2] as depending on the difficulty for a reverse-engineer to recognize that cloned functions are identical.

1.7 Ordering Transformations

Programmers tend to write logically related things close in the source code. This provides clues to a reverse engineer. Thus, we randomize the placement of items as much as possible. In some cases it is trivial, and in other, we have to determine which re-orderings are legal and will not affect the behavior.

1.7.1 Shuffling Basic Blocks

After applying control-flow hiding transformations such as Table Interpretation, the position of basic blocks can still give clues of the real control flow. To avoid that, we can shuffle all basic blocks within a function. This obfuscation itself has zero potency and low resilience but can be helpful when used with other transformations.

1.8 Data Transformation

1.8.1 Strings Encryption

Strings usually offer valuable knowledge or clues to a reverse-engineer. Even worse is that strings can be usually obtained with zero effort. By encrypting them, we make access to them more difficult. The process of encryption is straightforward, we encrypt all strings in the binary and then, every-time when we want to use any string we simply decrypt it on demand.

The binary has to contain at least one copy of the decryption algorithm. Thus the encryption leads to increase of binary size. The decryption process brings additional computing effort.

1.9 Summary

In this chapter, we introduced obfuscation transformation, evaluation metrics to assess potency, resilience and performance costs. We described basic concepts such as basic blocks, control flow graph to represent a function. We mentioned opaque predicates and aliasing predicates whose static analysis was proved to be NP-hard.

Then we went through categories of obfuscation transformations — layout, control, and data and for each category, we mentioned concrete examples such as Inserting Dead Code, Inlining, Virtualization, etc.

Existing Tools

This chapter will present existing obfuscation tools. The profoundly most obfuscators exist for Java/C++, but other languages are very sporadic. As we are going to create an obfuscator based on LLVM (will be mentioned in the next chapter), we are mainly interested in compiled languages supported by this tool. Thus, we only consider tools for C/C++, Swift, Obj-C, and additionally assembly and Portable Executable (PE).

2.1 Commercial Tools

2.1.1 VMProtect

VMProtect¹ obfuscates executables in PE format, which limits the usage to Windows platform. The tool offers two methods for obfuscation:

- “Mutation” that replaces source code instructions with sequences of instructions that produces the same result but are much harder to analyze. This method also hides calls to third-party libraries.
- Virtualization mentioned in 1.5.6.

Due to virtualization, the tool seems very powerful. The only disadvantage is that it supports only Windows platform.

2.1.2 Themida

Themida² obfuscates PE executables. It has the following important obfuscation techniques:

- Virtualization,

¹<http://vmpsoft.com/products/vmprotect/>

²<https://www.oreans.com/themida.php>

2. EXISTING TOOLS

- insertion of dead code.

This tool also supports virtualization which also makes it powerful. The tool is available for years and has many consumers, so their implementation of virtualization must be stable and efficient.

2.1.3 PELock/Obfuscator

PELock/Obfuscator³ works on the assembler level that supports Microsoft Macro Assembler (MASM) syntax. The tool offers the following obfuscation methods:

- change of code execution flow,
- mutation of original instructions into a series of equivalent opcodes,
- hide of direct calls to functions,
- insertion of dead code.

Due to limitations of assembly language to MASM, it effectively supports only Windows platform. The tool is available as a Windows application, an online interface or a web API.

2.1.4 CXX-OBFUS

CXX-OBFUS⁴ is an obfuscation tool for C/C++ that works on the source code level. The tool is cross platform and provides the following selected obfuscation techniques:

- replacing identifiers with random strings,
- replacing numerical constants with expressions that are harder to understand e.g., 232 with (0x14b6+2119-0x1c15),
- replacing characters in strings with hex escapes e.g., “cust” with “\x63\x75\x73\x74”,
- renaming source code files.

Unfortunately, this tool does not change the logic and these transformations will not help much against reverse engineering the compiled binary, as the most of the changes would not propagate into the binary anyway or could be easily simplified (replacing numerical constants) by a compiler.

³<https://www.pelock.com/products/obfuscator>

⁴<http://stunnix.com/prod/cxxo/>

2.2 Open-source Tools

2.2.1 Tigress C Obfuscator

Tigress C Obfuscator⁵ was created as a research project by (Collberg — co-author of [2]) and others. It supports only C language and works on the source code level.

The tool supports three major transformations:

- Virtualization mentioned in this thesis (creates a different instruction set for each method),
- transform function that generates its machine code at runtime (Jitting⁶),
- transform function that continuously modifies its machine code during runtime (JitDynamic⁷).

And three additional transformations:

- Control-Flow Flattening mentioned in 1.5.4,
- Aggregation Transformations mentioned in 1.6,
- Bogus Control Flow mentioned in 1.5.5.

The tool seems very powerful, and its authors even offer reverse engineering challenges⁸ to deobfuscate binaries obfuscated by this tool. The only disadvantage is that it supports only C language. Seeing the major transformations implemented for other languages or possibly using the LLVM platform (will be mentioned in the next chapter) would be amazing. It also seems still being maintained, as the last version was published in July 2018.

2.2.2 Swift Shield

Swift Shield⁹ is an obfuscation tool for iOS applications (Swift/Obj-C languages). The tool works on the source code level and automatically produces conversion maps, so we can easily decode crash logs. Unfortunately, it is only changing identifiers, and the logic remains untouched.

⁵<http://tigress.cs.arizona.edu/index.html>

⁶<http://tigress.cs.arizona.edu/transformPage/docs/jitter/index.html>

⁷<http://tigress.cs.arizona.edu/transformPage/docs/dynamic/index.html>

⁸<http://tigress.cs.arizona.edu/challenges.html>

⁹<https://github.com/rockbruno/swiftshield>

2.2.3 iOS Class Guard

iOS Class Guard¹⁰ is an obfuscation tool for iOS applications that supports Obj-C language. It works on the compiled version of an application, and same as Swift Shield, it only obfuscates identifiers (class, protocol, property and method names). The tool produces conversion maps in a JSON format.

2.2.4 Obfuscator-LLVM

Obfuscator-LLVM was initiated in 2010 to provide increased software security through code obfuscation and tamper-proofing. It works on the LLVM IR level (mentioned in subsection 3.1.3). Thus it is compatible with all languages and platforms supported by the LLVM platform. The tool supports the following methods: Instructions Substitution, Bogus Control Flow and Control Flow Flattening.

Instructions Substitution replaces standard library operators like addition, subtraction or boolean by an equivalent but much more complicated instructions.

For instance it replaces $a = b - (-c)$ with $r = rand()$, $a = b - r$, $a = a + b$ and $a = a + r$. This obfuscation does not add much security, as it can be easily removed by re-optimizing the generated code. Anyway, by using pseudo random generator seeded with different values, instructions substitutions bring diversity into the produced binary.

Unfortunately, the project does not seem to be actively maintained, has many many unsolved issues on GitHub and there is no new code since 2017.

2.2.5 Petráček's Obfuscator

Petráček's [15] tool is based on LLVM platform. The implemented methods are Inlining, Outlining, Interleaving, Splitting Blocks, Bogus Control Flow and Table Interpretation.

The implemented Table Interpretation seems more advanced than the same transformation in Obfuscator-LLVM, as it has implemented opaque predicates, and adding invalid states to the dispatcher. It also increases the resilience by not storing the direct identifier of the next BB but only a difference between identifiers of current's and successor's BB. The dispatcher is confusing a potential reverse engineer by using indirect jumps (*indirectbr*) that can jump outside a function. For this reason, IDA (disassembler) is often unable to correctly recognize an obfuscated function.

¹⁰<https://github.com/Polidea/ios-class-guard>

2.2.6 Šíma's Obfuscator

Šíma's [17] tool is also based on the LLVM platform. The tool offers Dead Code, Interleaving and Table Interpretation.

Dead Code transformation uses conditional jumps based on opaque predicates to add dead and irrelevant code. The transformation provides CLI parameters for granularity control.

2.3 Summary

The selected commercial tools mostly support Windows Platform and provide only little details about used transformations. We can understand their unwillingness to reveal the details to reverse-engineers, but this decreases our capability to evaluate these tools. The most interesting tools seem to be VM-Protect and Themida due to virtualization.

The selected open-source tools support C/C++/Swift/Obj-C. The tools for iOS provide mainly layout transformations and do not seem to be very powerful. The most powerful tool seems to be Tigress C Obfuscator, as it supports advanced transformations such as Jitting or JitDynamic that are far beyond our capabilities. Unfortunately, it supports only C language and requires a lot of manual configuration. Obfuscator-LLVM is based on LLVM, and it is very similar to the tool we are going to implement but does not offer as good transformations as we want to create and it does not seem to be actively maintained.

Petráček's and Šíma's tools seem to be more powerful than Obfuscator-LLVM, even Obfuscator-LLVM is probably more stable and tested, because it was made available for mass usage. As they are both based on the LLVM platform, we will select the most advanced transformations from each tool and bundle them together with transformations created in this thesis to create a tool that offers a complex set of transformations.

Design

In this chapter, we will be talking about the design of the proposed obfuscation tool. The chapter begins with the introduction of the LLVM Framework, the essential building block of our obfuscator. Then we will describe a design of several obfuscation techniques that are about to be implemented. Finally, we will mention CLI parameters and pragmas to control the obfuscation process.

3.1 LLVM Framework

LLVM [8] began as a research project at the University of Illinois with the aim to provide a modern compilation strategy for any programming language. Currently, LLVM is an umbrella project that consists of many subprojects such as LLVM Core, Clang (C/C++/Obj-C compiler) and many more.

3.1.1 Classical Compiler Design

The most common design for a static compiler consists of the front-end, optimizer, and back-end [8], as shown in fig. 3.1. The front-end parses the source code, checks it for errors and converts it to a language-specific Abstract Syntax Tree. Then it can be converted into an internal representation used by optimizer and back-end. The optimizer is doing various transformations to improve the efficiency such as reducing redundant computations, and it is usually more or less independent on a source language and target architecture. The back-end maps the code onto the target instruction set.

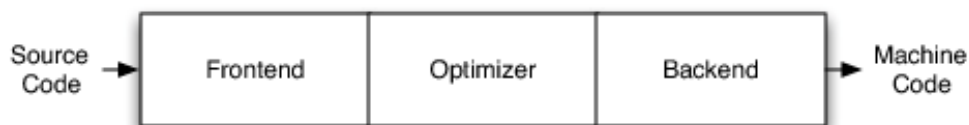


Figure 3.1: Classical Compiler Design [8]

3. DESIGN

The main pro of this design is that a compiler can support multiple languages and target architectures. As we can see in fig. 3.2, adding support of additional language only requires implementing a new front-end, but the rest can be reused. If these parts were not separated, we would need to start every-time from scratch.

If a compiler serves a broader audience, it usually helps to make it better, more reliable, to introduce better optimizations, etc. The excellent example of this is GCC.

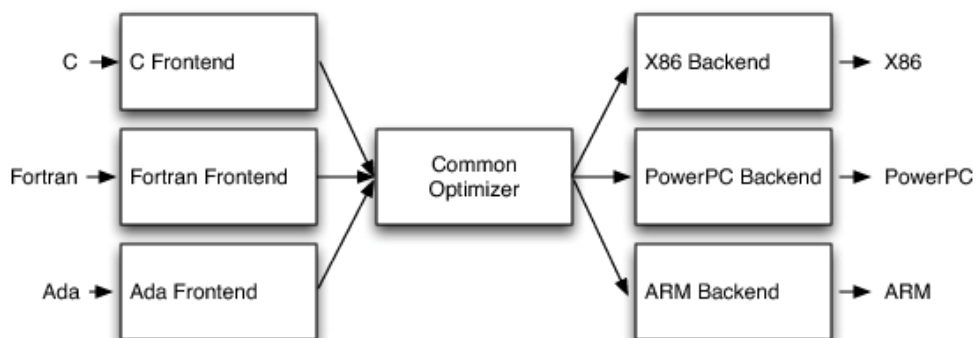


Figure 3.2: Design of a compiler with multiple front-ends and back-ends [8]

3.1.2 LLVM's Implementation of Three-Phase Design

In LLVM's implementation (see fig. 3.3), a front-end is responsible for processing a source code and converting it to LLVM IR. Then LLVM Optimizer improves the efficiency of code, and in the end, the code is sent into a back-end to produce native machine code.

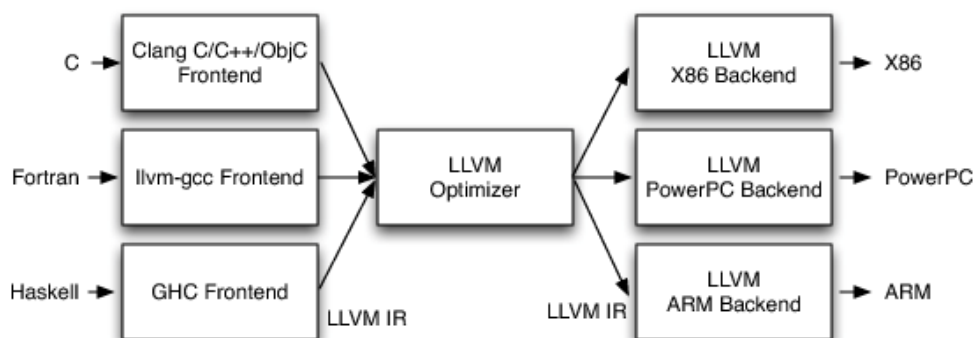


Figure 3.3: LLVM's implementation of Three-Phase Design [8]

<pre> int sum(int n) { if(n == 0) return n; return n + sum(n-1); } </pre> <p>(a) C language</p>	<pre> define i32 @sum(i32 %n) { entry: %cmp = icmp eq i32 %n, 0 br i1 %cmp, label %if.then, label %if.end if.then: ret i32 %n if.end: %sub = sub nsw i32 %n, 1 %c = call i32 @sum(i32 %sub) %add = add nsw i32 %n, %c ret i32 %add } </pre> <p>(b) LLVM IR</p>
---	--

Figure 3.4: C function represented in LLVM IR. Note: some instructions were wrapped for a better readability.

3.1.3 LLVM IR

LLVM IR [8] is the most significant part of LLVM design, which is a code represented inside the compiler. It is defined as a first class language with the aim to represent any high-level language ideas and has many features such as lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, aggressive restructuring transformations, etc.

LLVM IR is Static Single Assignment (SSA) language, strongly typed (e.g., uses **i32** for a 32-bit integer), has an infinite set of virtual registers named with **%** prefix. It supports linear sequences of instructions like add, compare, branch, etc. and also supports labels. Instructions are in three address form; they take some inputs and produce a result in a different register.

Due to efficiency, LLVM IR can be represented in three equivalent forms [10]: an in-memory compiler IR, an on-disk bitcode representation for a JIT compiler (has a **.bc** extension) and a human readable assembly language representation (has a **.ll** extension).

In figure 3.4 we can see that LLVM IR looks like some sort of compromise between an assembly language and procedural language.

3. DESIGN

```
define i32 @nsum(i32 %n) {
entry:
    %retval = alloca i32
    %n.addr = alloca i32
    store i32 %n, i32* %n.addr
    %0 = load i32, i32* %n.addr
    %cmp = icmp eq i32 %0, 0
    br i1 %cmp,
        label %if.then,
        label %if.end

if.then:
    %1 = load i32, i32* %n.addr
    store i32 %1, i32* %retval
    br label %return

if.end:
    %2 = load i32, i32* %n.addr
    %3 = load i32, i32* %n.addr
    %sub = sub nsw i32 %3, 1
    %c = call i32 @nsum(i32 %sub)
    %add = add nsw i32 %2, %c
    store i32 %add, i32* %retval
    br label %return

return:
    %4 = load i32, i32* %retval
    ret i32 %4
}

define i32 @nsum(i32 %n) {
entry:
    %cmp = icmp eq i32 %n, 0
    br i1 %cmp,
        label %if.then,
        label %if.end

if.then:
    br label %return

if.end:
    %sub = sub nsw i32 %n, 1
    %call = call i32
        @nsum(i32 %sub)
    %add = add nsw i32 %n,
        %call
    br label %return

return:
    %rv.0 = phi i32
        [ %n, %if.then ],
        [ %add, %if.end ]
    ret i32 %rv.0
}

(b) Using PHI instruction to select
a value
```

(a) Value is stored in memory

Figure 3.5: A comparison of two approaches for obtaining value from preceding BBs. Note: code was optimized for better readability.

3.1.3.1 Static Single Assignment (SSA)

As we already mentioned, LLVM IR is an SSA language, which means that a virtual register can have assigned only one value that cannot be changed. This is not a case for memory operations, which are not in the SSA form [14]. For the most programming languages, it is essential to change the value of a variable frequently. This introduces a problem for the SSA form in case of a value coming from one of the preceding BBs [9].

As we can see in fig. 3.5 LLVM IR addresses this issue either by storing

value in memory (load and store instructions) or by using PHI instruction that assigns value based on a which preceding BB was executed. Both forms are frequently used and LLVM IR has built-in transformations for the conversion between them: “mem2reg” — promoting memory to register and “reg2mem” — demoting register to memory.

3.1.3.2 Exception Handling

LLVM IR has three exception-handling related instructions¹¹:

- **landingpad** is used to specify that a basic block is an exception handling block — LLVM documentation refers to this block as a landing pad. It occurs at the beginning of BB and can be preceded only by *PHI* instruction.
- **invoke** is similar to a *call* instruction but it takes labels of two BBs, the first one is used if a function returns normally and the second one is used for exception propagation. In contrast of *call*, *invoke* is a terminator instruction, and it has to be last in a BB.
- **resume** is used to resume propagation of an exception which was earlier caught by combination of *invoke* and *landingpad* instructions.

LLVM IR does not provide any instruction to throw an exception. This is left for front-end, e.g., in C++ is throwing achieved by calling `__cxa_throw`.

3.1.4 LLVM Pass Framework & LLVM API

The most significant part of LLVM Framework is LLVM Pass Framework, as it offers a rich C++ API to perform transformations or analyses on LLVM IR — referred to as *passes*.

Each element of LLVM IR (global variable, module, function, BB, etc.) is represented as a class that provides methods to access and modify it. In order to use LLVM, we need to inherit from *Pass* class and create our own *pass*¹².

The API also offers various utilities to perform complicated manipulations such as:

- **ReplaceInstWithInst** replaces an instruction with another instruction and all users of the deleted instruction are updated to use the new one.

¹¹<https://llvm.org/docs/ExceptionHandling.html>

¹²<https://llvm.org/docs/WritingAnLLVMPass.html>

- **Value::replaceAllUsesWith** *Value* is a subclass of *Instruction* and represents *Instruction's* result and can be used as operand. Method *replaceAllUsesWith* updates all users of a *Value* to use a new one.

3.1.5 LLVM CLI Basic Usage

In this subsection we will describe several essential command line tools¹³ that are inevitable to use the LLVM.

- **clang** is a C/C++/Objective-C compiler and part of the LLVM project. The tool can also transform an input code to LLVM IR either in LLVM assembly language (human readable) or LLVM bitcode format.

```
# C -> LLVM assembly
$ clang -emit-llvm -S example.c -o example.ll
# C -> LLVM bitcode
$ clang++ -emit-llvm -c example.cpp -o example.bc
```

```
# C++ -> LLVM assembly
$ clang++ -emit-llvm -S example.cpp -o example.ll
# C++ -> LLVM bitcode
$ clang++ -emit-llvm -c example.cpp -o example.bc
```

```
# LLVM assembly in a more readable format
$ clang -emit-llvm -fno-discard-value-names -S
  ↪ example.cpp -o example.ll
```

- **llvm-as** is the LLVM assembler which transforms LLVM IR in LLVM assembly language to LLVM bitcode format.

```
$ llvm-as example.ll -o example.bc
```

- **llvm-dis** is the LLVM disassembler, which transforms LLVM IR in LLVM bitcode format to LLVM assembly language.

```
$ llvm-dis example.bc -o example.ll
```

- **llvm-link** is the LLVM linker links two or more LLVM bitcode files together.

```
$ llvm-link example1.bc example2.bc example3.bc -o
  ↪ example123.bc
```

¹³<https://llvm.org/docs/CommandGuide/>

- **lli** executes a program in LLVM bitcode format using a just-in-time compiler or interpreter.

```
$ lli example.bc
```

- **llc** is a compiler that takes an input either in LLVM assembly language or in LLVM bitcode format and transforms it into assembly language for a specified architecture.

```
$ llc example.bc -o example.s
```

- **opt** is the modular LLVM optimizer and analyzer. It accepts input either in the LLVM assembly language format or in the LLVM bitcode format, performs the selected optimizations or analyses on it, and returns an optimized file or analysis results. It is the most important tool for us, as we will use this tool to run our obfuscator.

Examples of built-in analyzers¹⁴:

- **basicaa**: basic alias analysis
- **da**: dependence analysis
- **instcount**: counts the various instruction types
- **loops**: natural loop information

Examples of built-in optimizers¹⁴:

- **constprop**: simple constant propagation
- **dce**: dead code elimination
- **inline**: function inlining
- **instcombine**: combine redundant instructions
- **licm**: loop invariant code motion
- **tailcallelim**: tail call elimination

```
$ clang -emit-llvm -S example.c -o example.ll
```

```
# analyze loops
$ opt -analyze -loops example.ll
```

```
# run dead code elimination
$ opt -dce -S example.ll -o example_dce.ll
```

¹⁴<https://llvm.org/docs/Passes.html>

<code>%a = alloca i32</code>	1 -> <code>%a = alloca i32</code>
<code>%b = alloca i32</code>	2 -> <code>%b = alloca i32</code>
<code>store i32 5, i32* %a</code>	3 -> <code>store i32 5, i32* %a</code>
<code>store i32 90, i32* %b</code>	4 -> <code>store i32 90, i32* %b</code>

(a) BB's *commands* (b) BB's *commands* with assigned opcodes

Figure 3.6: An example of assigning opcodes.

3.2 Transformation Passes

In this section, we will discuss the design of our implementation.

3.2.1 Lightweight VM Pass

This pass will destroy a function's CFG by transforming its BBs into a virtual machine with custom opcodes. To describe this pass, we need to define a new term *LLVM Assembly Command* or shortly *command*, which will stand for instruction with operands and assignment to a virtual register (if applicable). For instance *alloca* is an instruction and `%a = alloca i32` is a command.

We see two available ways of implementing virtual machines based obfuscation in LLVM. The first one is transforming the whole function into VM so that the former control flow will completely disappear. This approach produces better resilience but also has downsides. The main downside is that not all BBs can be easily obfuscated, as they may throw an exception, or for a variety of another reason. The second approach is BBs based obfuscation. Sometimes the user may want to go for a trade-off between the performance and quality of the obfuscation by obfuscating only some portion of function's BBs. This can be done easily with this approach; thus we selected it for the implementation.

3.2.1.1 Opcodes

We will assign a random opcode to every assembly command and move these commands into VM interpreter. We can see in Figure 3.6 that commands are considered uniquely, and instructions with different operands have assigned different opcodes. This is due to the limitation of LLVM IR, which does not allow to set instruction's operands or virtual registers dynamically. As a result, the VM interpreter will be more robust, and this will result in some performance penalty caused by switch implementation. From the obfuscator's quality point of view, the obfuscated function should be harder to reverse engineer, as one instruction will have multiple opcodes and it will be much harder to write a disassembler to reverse this transformation.

3.2.1.2 VM Interpreter

The VM interpreter is a combination of while and switch. It is located inside an obfuscated function, and each obfuscated function has its interpreter. The interpreter loads an opcode from array assigned in the obfuscated BB and selects appropriate command according to the opcode.

3.2.1.3 Summary

We would like to sum up the all important features of the Lightweight VM Pass:

1. the obfuscation will be conducted on BBs of a particular function,
2. user will be able to set the maximum number of obfuscated BBs,
3. VM Interpreter will be stored inside an obfuscated function,
4. for assigning opcodes, we will consider every command to be unique.

This technique seems very potent & resilient at the expense of slowdown. The roughly estimated slowdown is around twelve times. Therefore, it should not be used on the entire application but only for the parts that require a higher level of protection (e.g., watermark, proprietary algorithm, an algorithm that implements security by obscurity, etc.).

3.2.2 String Encryption Pass

Strings are trivially obtainable and can provide significant clues to a reverse-engineer, which give us a strong motivation for creating this transformation.

There are two main requirements for this pass, the first one is quick decryption to keep the obfuscation penalty as low as possible and the second one is the minimal increase of binary size.

Stream ciphers are very efficient, but also prone to vulnerabilities [5]. Therefore, strings will be encrypted using a block cipher AES-256 in CBC mode. To achieve fast decryption, we will use the optimized implementation of AES [16]. The optimized AES uses bigger tables with the size around 6KB, twelve times more than tables used in the standard AES implementation, which is a tax of the faster decryption.

We will encrypt all strings up to two-dimensional arrays of chars (or one-dimensional arrays of strings). Three and more dimensional arrays of chars are very rare, and their encryption is tricky, so we leave them from the encryption process. Wide strings will also be supported, and we will obtain the length of wide chars from metadata in LLVM IR.

3.2.2.1 Increasing resilience

Essentially, it is important to shadow information about the used cipher. The most typical things that can reveal AES are its precomputed tables. We will try to hide them by creating multiple variations of these tables, where each variation will contain values XORred with a different random value.

Calls to decryption algorithm can also reveal information about the used cipher. We tried inlining the whole decryption algorithm for every decryption, but it led to a significant increase of binary size. In the case of 1000 strings, it went from 56KB to 5MB; thus we had to go for a trade-off. We will create a function for every variation of AES tables, and this function will have inlined the whole decryption algorithm. In the decryption process, we will randomly choose one of these functions. User will be able to control the number of decryption functions by selecting from one up to twelve decryption functions.

3.2.2.2 Summary

We want to sum up the all important features of the String Encryption Pass:

1. strings will be encrypted using AES-256 in CBC mode,
2. support of standalone strings and up to two-dimensional arrays of characters,
3. wide strings support,
4. for the decryption we will use randomly one of several decryption functions with inlined decryption algorithm, where each function will use a unique variation of AES tables *XORed* with a random number.

3.2.3 Shuffle BB Pass

A position of basic blocks can provide significant clues to a reverse engineer. Therefore, we will create a pass to randomize positions of all BBs except the entry one. This pass is entirely legal and will not cause any issues, as all references to BBs will stay untouched.

3.2.4 Improved Function Interleaving Pass

Function Interleaving implemented in [15] merges only two functions together. We can run it multiple times, but this will bring up additional parameters. Therefore, we see there a motivation for improving it so that it can merge an unlimited number of functions.

The second improvement will be adding dead code to add more confusion. The dead code will consist of bodies from existing functions.

The excellent advantage of the current implementations is that it uses opaque predicates. This will not be a part of the new implementation. Thus, we decided to go for a compromise. There will be CLI parameters to control how many functions can be interleaved together and how many dead functions can be added. If these parameters are set to two and zero (respectively), then the algorithm will fully fall back to the implementation with opaque predicates.

3.2.5 Included Passes

Our tool will also contain the following passes from Petráček's [15] and Šíma's [17] theses:

1. Bogus Control Flow, Inlining, Opaque Predicates, Outlining, Table Interpretation and Split Blocks from [15],
2. DeadCode from [17].

The reason for bundling these passes together with obfuscation transformations created in this thesis is that we want to create an obfuscation tool providing a complex set of transformations. It is also more convenient for the user to have everything in one tool without the need to build the LLVM with multiple obfuscation tools to achieve the same result.

3.3 Grain Control of Obfuscation Passes

Ability to control obfuscations is the essential requirement of obfuscator's users. We will implement CLI parameters that will be applied to the whole module and pragmas applied to a specific function or variable.

The priority of parameters will be in a natural way. CLI parameters will override default parameters, and pragmas will override parameters for a specific function or variable. We will implement grain control also to bundled passes created in [15, 17].

3.3.1 Pragmas

Implementing custom pragmas would require customization of Clang/LLVM code to parse pragmas and propagate them to LLVM IR. Therefore, we decided to use annotations, as they are naturally supported by Clang and propagated to LLVM IR. We can see an example of pragmas in Figure 3.7. Pragmas with annotations do not look so good as custom pragmas, but it is always better to use something supported by framework than customizing it, as it brings additional effort to end users (they would have to patch LLVM and compile it manually) and decreases compatibility with future's versions.

```
// annotate variable
#pragma clang attribute push (__attribute__((annotate(
    "param=value"
))),
    apply_to = variable)

int foo;
#pragma clang attribute pop

// annotate function
#pragma clang attribute push (__attribute__((annotate(
    "param=value"
))),
    apply_to = function)

void foo();
#pragma clang attribute pop
```

Figure 3.7: An example of pragmas. Note: pragmas are one-liners, the code was wrapped for better readability.

3.4 Summary

In this chapter, we introduced a three-stage design, LLVM Framework, LLVM IR and LLVM Pass Framework. We discussed SSA, Exception Handling, LLVM API, and mentioned essential CLI commands to work with LLVM. We created a design of our obfuscation tool consisting of the following obfuscation transformations: Lightweight VM, String Encryption and Improved Function Interleaving. In the end, we will include selected obfuscation passes from these [15, 17] in our tool. All passes will support parameters to gain control obfuscation process.

Implementation

This chapter deals with the implementation details of the created obfuscation tool and in several cases provides concrete examples of transformed code. We will also mention the all significant struggles we encountered during the implementation. For instance, in the case of Lightweight VM Pass, we had to deal with switch and allocation instructions. And String Encryption Pass brought issues with Constant Expressions. We will also mention cases, where we implemented CLI-parameters or pragmas for a granularity control of the obfuscation process.

4.1 Lightweight VM Pass

VM pass is performed on BBs of a particular function. At the beginning of the transformation, we have to get rid of *PHI* instructions by using *reg2mem* pass, as *PHI* instructions would not work well with VM.

We assign an opcode to every *command* in a BB and move all *commands* to a VM Interpreter. For every *command*, we create a separate BB in the VM interpreter. Then we create an array with all opcodes assigned to *commands* from a BB. In the end, the obfuscated BB will contain only the assignment of the array with opcodes to a shared opcode-array pointer used by the VM Interpreter and jump to the VM Interpreter.

The VM interpreter iterates over the opcodes array assigned in a BB that started the interpreter. According to the opcode value, it selects a BB containing a requested command. If the command is not a terminating instruction, the BB jumps back to the body of VM interpreter. The last instruction in every BB should be a terminating one, so the VM interpreter does not check the size of operands array. In Listing 4.1 we can see an example of C function, its representation in LLVM IR and VM obfuscated version.

4. IMPLEMENTATION

```
1 // C function
2 int fn() {
3     int a = 5;
4     return a;
5 }
6
7 // non-obfuscated function
8 define i32 @fn() #0 {
9     entry:
10    %a = alloca i32, align 4
11    store i32 5, i32* %a, align 4
12    %0 = load i32, i32* %a, align 4
13    ret i32 %0
14 }
15
16 // obfuscated function
17 @opcodes = private constant [4 x i32] [i32 609348, i32 609350, i32
18     609349, i32 609351]
19 define i32 @fn() #0 {
20     entry:
21     %opcodesPtr = alloca i32*
22     %opcodesGVCast = bitcast [4 x i32]* @opcodes to i32*
23     store i32* %opcodesGVCast, i32** %opcodesPtr, align 4
24     br label %VMInterpreter
25
26 VMInterpreter:
27     %i = alloca i32, align 4
28     store i32 0, i32* %i, align 4
29     %loadedOpcodePtr = load i32*, i32** %opcodesPtr
30     br label %VMInterpreterBody
31
32 VMInterpreterBody:
33     %loaded_i = load i32, i32* %i
34     %increased_i = add i32 %loaded_i, 1
35     store i32 %increased_i, i32* %i, align 4
36     %opcodesIdx = getelementptr i32, i32* %loadedOpcodePtr, i32 %
37         loaded_i
38     %loadedOpcode = load i32, i32* %opcodesIdx
39     switch i32 %loadedOpcode, label %VMInterpreterBody [
40         i32 609348, label %VMInterpreterBB
41         i32 609349, label %VMInterpreterBB1
42         i32 609350, label %VMInterpreterBB2
43         i32 609351, label %VMInterpreterBB3
44     ]
45
46 VMInterpreterBB:
47     %a = alloca i32, align 4
48     br label %VMInterpreterBody
49
50 VMInterpreterBB1:
51     %0 = load i32, i32* %a, align 4
52     br label %VMInterpreterBody
```

```

53 VMInterpreterBB2:
54   store i32 5, i32* %a, align 4
55   br label %VMInterpreterBody
56
57 VMInterpreterBB3:
58   ret i32 %0
59 }

```

Listing 4.1: An example of VM obfuscation

4.1.1 Switch Implementation Details

The most appropriate way of implementing switch inside the VM Interpreter is LLVM’s *switchinst*. The backend compiler will later use the most appropriate way for selecting a branch, and in the best case, it will use jump tables which are supposed to be the most efficient.

Unfortunately, using *switchinst* is sometimes leading to issues (mainly when used with non-succeeding opcodes); thus we decided also to implement our solution. Our solution is based on the divide-and-conquer approach with $\mathcal{O}(\ln n)$ complexity, where n is the number of commands in the VM interpreter. This solution is way slower than the most efficient way that can be used for implementing *switchinst* (jump tables) but does not lead to issues with non-succeeding opcodes.

We implemented a parameter, and a user can select either *switchinst* or our implementation.

4.1.2 Allocation Instructions

Moving allocation instructions into the VM Interpreter sometimes leads to issues, as the compiler has zero information of their life-cycle. Therefore, we decided to move these instructions to the entry BB and keep them away from the obfuscation.

We also implemented a parameter to enforce the obfuscation of these instructions.

4.1.3 Random Opcodes

We use two types of random opcodes: random non-succeeding opcodes and succeeding opcodes. Succeeding opcodes are completely random numbers, while in case of non-succeeding opcodes the first opcode starts with a random number and the next opcodes are the following numbers to the first one. In both cases *commands* in a particular function have randomly assigned opcodes. The motivation behind this is that if we use non-succeeding opcodes then *switchinst* is sometimes leading to issues. Also, the compiler cannot efficiently use jump tables for implementing switch with non-succeeding opcodes.

We decided to provide a user a grain control over the VM obfuscation. User can choose between succeeding opcodes and non-succeeding ones or between *switchinst* and the custom switch implementation. The most appropriate combinations are non-succeeding opcodes with *switchinst* and succeeding opcodes with the custom switch implementation. The first option should result in better performance.

4.1.4 The maximum number of obfuscated BBs

Since this pass brings a significant performance penalty, the user may want to choose a trade-off between the quality of obfuscation and performance. We implemented a parameter to set the maximum number of BBs obfuscated in a particular function.

4.1.4.1 Ignored BBs

We decided not to obfuscate some types of basic blocks. Either they have too many corner cases, or their obfuscation is not worth the performance penalty. List of such types:

1. exception handlers,
2. BBs that contains a *PHI* instruction (reg2mem does not always remove all of them),
3. BBs with less than 6 selected for obfuscation.

4.2 String Encryption Pass

Surprisingly, the implementation of this pass was the most challenging part of this thesis. The main issue arises from the behavior of Constant Expressions in LLVM API. We will cover this issue in the following subsection.

The process of encryption can be described easily; we select all standalone strings and strings in 2D arrays that are appropriate to be encrypted and encrypt them. We also implemented the support for wide strings. In the end, we find all places where the encrypted string is being accessed and add a there a decryption function.

4.2.1 Constant Expressions

Constant Expression is a constant initialized with an expression using other constants. In the context of String Encryption, we mainly focus on expression called *getelementptr*. See an example of such expressions in Figure 4.1.

Constant expressions in LLVM are implemented in a way that if one expression is used multiple times, only one instance of this expression is created

and shared everywhere. While the intention of making the LLVM Pass Framework more performance efficient is clear and completely understandable, it also brings issues to programmers using this API. The undesirable consequence of this is that if we alter a constant expression, then we have to keep in mind that it will be changed everywhere, and we have to fix all places of occurrence.

As an example, we can consider the following case. We have a string that is used at multiple places, let us say in a 2D and 3D arrays. If we obfuscate string in the 2D array and change the reference, the reference to the string will also be updated in the 3D array, which we are not going to obfuscate.

4.2.2 Duplicating strings

To address the issue with constant expressions, we duplicate strings if they are used in multiple arrays and fix references, so every array uses a unique copy of a string. In Figure 4.1 we can see an LLVM IR representation of the case when one string is used in multiple arrays. String *Lorem* is standalone but also used in the 2D array, string *Ipsum* is used in the both 2D and 3D arrays. *GetElementPtr* is a constant expression, and in the LLVM API, the same constant expressions are instanced only once and shared between all occurrences.

4.2.3 Wide strings

For the support of wide strings, we obtain wide char length from the metadata in LLVM IR. The encrypted format of the string is the same for all strings, regardless of their wideness, and they are stored in an array of bytes. Since our decryption function only works with arrays of bytes, we *bitcast* an array of decrypted string to have the correct byte-width.

4.2.4 Arrays

There are also cases when a string in an array is accessed dynamically (e.g., in loops), so at the compile time, we have no idea which string will be used. To address this issue, we have to use the same key for all strings inside the array, and we also store sizes of decrypted strings in a separate array.

4.2.5 IV & Decryption Keys

Every string is encrypted using a unique random key, except arrays, where all strings in a particular arrays share the same key. Since we use a random key, we believe there is no need for a random IV, and we decided to use hard-coded & shared IV for all encryptions.

Hardening Decryption Keys AES uses 32-bytes long keys for decryption. This can reveal information about used cipher to a reverse-engineer. To

4. IMPLEMENTATION

```
// C language
const char str[] = "Lorem";
const char* _2D[] = {
    "Lorem", "Ipsum"
};
const char* _3D[2][2] = {
    { "Lorem", "Ipsum" },
    { "Dolor", "Sit" },
};

// LLVM IR
@.str = private unnamed_addr constant [6 x i8] c"Lorem\00"
@.str.1 = private unnamed_addr constant [6 x i8] c"Ipsum\00"
@.str.2 = private unnamed_addr constant [6 x i8] c"Dolor\00"
@.str.3 = private unnamed_addr constant [4 x i8] c"Sit\00"
@2D = global [2 x i8*] [
    i8* getelementptr inbounds (
        [6 x i8], [6 x i8]* @.str, i32 0, i32 0
    ),
    i8* getelementptr inbounds (
        [6 x i8], [6 x i8]* @.str.1, i32 0, i32 0
    )
]
@3D = global [2 x [2 x i8*]] [
    [2 x i8*] [
        i8* getelementptr inbounds (
            [6 x i8], [6 x i8]* @.str, i32 0, i32 0
        ),
        i8* getelementptr inbounds (
            [6 x i8], [6 x i8]* @.str.1, i32 0, i32 0
        )
    ],
    [2 x i8*] [
        i8* getelementptr inbounds (
            [6 x i8], [6 x i8]* @.str.2, i32 0, i32 0
        ),
        i8* getelementptr inbounds (
            [4 x i8], [4 x i8]* @.str.3, i32 0, i32 0
        )
    ]
]
```

Figure 4.1: An example of strings used in multiple arrays. Note: the code was optimized for better readability.

shadow this information, we will randomly add a fill-in in the form of one up to four additional bytes to the decryption key.

4.2.6 Storage for Decrypted Strings

No less important is to discuss where to store decrypted strings. For selecting proper storage, we have to consider the following situations. In most cases, we use *alloca* to allocate memory on stack to store a decrypted string. However, if a function returns a pointer to a *char/wchar_t* array that is directly or indirectly related to the decrypted string, there is a possibility that the function returns the decrypted string and in this case, we cannot use the memory allocated on the stack.

Therefore, we have to decrypt string to a global array. The global array is created only in the case that there is at least one string that needs to be decrypted there. Every string decrypted to the global array has its exclusive position within the array, and once is decrypted, it never gets never removed. This approach decreases the resilience, but we have to realize that the decryption to a global array will be used only in minor cases.

4.2.7 Decryption Functions

One of the ways how to hide information about AES is inlining decryption code. Tables used in AES also reveal a lot of information, so for each decryption function, we use unique tables *XORed* with some random value.

Decryption algorithm can reveal significant information about used cipher, and this can help a reverse-engineer to build an automatic tool to undo this obfuscation. Also, AES tables provide significant clues.

To make this process harder, we created decryption functions with inlined all decryption code and *XORed* tables with a random value — to make their identification more difficult. Each decryption function uses dedicated *XORed* tables and a user can specify how many (up to twelve) decryption functions he wants to use. For the decryption process, one of these functions is randomly selected. Using more decryption functions increases a binary size, as decryption function and dedicated *XORed* takes some space, together roughly around 8 KB.

4.3 Improved Interleaving Pass

We improved the Interleaving Pass created in [15] to be able to interleave an unlimited number of functions together. From the implementation point of view, we take one function, and for every other function with the same return type, we count how many arguments will have the interleaved function. From this list sorted by the number of arguments we take first *n* functions (depends on a parameter) and interleave them.

This is not the best way how to select functions for interleaving, as they may have mutually incompatible arguments, which will increase the number of arguments. The proper way of selecting is subset selection, which was proved to be NP-hard. It can be either implemented by trying all combinations or using heuristics.

The second improvement is adding dead code. We find a function that will not change the number of arguments, and we add it to the new interleaved function in the same way as we do in case of regular interleaving. However, in this case, we keep using the function that was added as a dead code to the interleaved function and will not update calls.

4.4 Summary

In this chapter, we discussed the implementation of our obfuscation tool. For the every implemented transformation, we mentioned how it was implemented, provided concrete examples and we were also talking about issues we encountered during the implementation. In case of Lightweight VM Pass, we mentioned why we decided to skip obfuscation of allocation instructions by default, what problems we had with switch and why we implemented two types of opcodes, succeeding and non-succeeding random opcodes.

During the implementation of String Encryption Pass, we had to deal with Constant Expressions and decided to duplicate strings to avoid potential issues. We also mentioned how we hardcoded decryption keys and what methods we used for storing strings after decryption.

In many cases, and also for included transformations from [15, 17] we implemented parameters for fine-grained control of the obfuscation process. We also implemented parameters for cases where we encountered problems and set them to have conservative default values. Therefore, the user still can try to achieve better quality at the expense of possible negative effects.

Evaluation

This chapter evaluates the created obfuscation tool. We will assess potency, resilience and the performance impact of the implemented obfuscation transformations. The performance impact of String Encryption Pass will be measured separately in order to obtain more representative results. We will also discuss the quality of Lightweight VM Pass. In the end, we will compare our tool with existing solutions.

5.1 Obfuscation Metrics

In this section, we will measure the obfuscation metrics of implemented transformations. The created tool also contains obfuscations from [15, 17]. However, the bundled transformations were already evaluated in the mentioned theses. Therefore, we will only evaluate transformations created in this thesis. It is necessary to remark that results of obfuscations may vary even in the case of the same transformations applied to the same program, as all obfuscations are randomized as much as possible.

5.1.1 Test Programs

For the evaluation we selected the following algorithms: AVL Tree¹⁵, AES¹⁶ and recursive QuickSort¹⁷. All algorithms were implemented in C language, and the source code was compiled by using clang with *O3* optimizations and disabled inlining; otherwise, the evaluation of Function Interleaving Pass would not be possible. We also increased a stack size, as we encountered problems in VM transformation applied to the recursive QuickSort. The virtualization transformation was always applied to all internal functions except the main function.

¹⁵<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

¹⁶<https://fastcrypto.org/front/misc/rijndael-alg-fst.c>

¹⁷<https://www.geeksforgeeks.org/quick-sort/>

5.1.2 Potency

Potency is defined in [2] as a measure of how difficult is to understand an obfuscated program compared to its non-obfuscated version. It is defined on a three-point scale: *low*, *medium* and *high*. For the evaluation, we picked the first two potency metrics defined in Table 1.1, changes of program length (μ_1) and changes of cyclomatic complexity (μ_2). In Tables 5.1 and 5.2 we can see potency metrics for selected programs.

Collberg et al. [2] estimated the potency of Function Interleaving as depending on used opaque predicates. In Improved Function Interleaving Pass we use simple switch without opaque predicates, and despite the quite good results in some metrics, we evaluated the potency to be *low*.

ShuffleBBs Pass was created to confuse a reverse-engineer, as a position of BB in a binary may provide him clues of the real control flow. This pass is not intended to be used standalone but as a combination with other transformations. Therefore, its potency is *low*.

Lightweight VM Pass is supposed [2] to provide a *high* potency. In the results, we can see that it provides better results for larger programs. AES is a quite huge algorithm with very high metrics, whereas QuickSort is a tiny algorithm and its metrics are much lower. Hence the potency is the same as was estimated in [2].

In the case of String Encryption Pass, we decided not to measure the potency at all, as there is no program containing the average number of strings of all programs and any result would be misleading. If a particular program contains zero strings, the result would be zero and vice-versa.

Obfuscation [settings]	AVL Tree		AES		QuickSort	
	Total	Max	Total	Max	Total	Max
VM [default]	3.04	3.45	4.10	4.55	2.97	2.30
VM [customSwitch, NSO]	4.53	5.41	6.63	7.83	3.98	3.91
VM [allocs]	3.37	3.79	3.89	4.26	2.85	2.08
ShuffleBBs [default]	1.00	1.00	1.00	1.00	1.00	1.00
Interleaving [default]	2.42	4.09	2.81	5.12	2.75	2.15
Interleaving [unlimited]	2.16	5.85	2.01	8.91	1.83	2.67
Interleaving [deadFns=0]	2.10	4.09	1.86	2.50	1.77	2.11

Table 5.1: Program length changes (μ_1) for selected programs. Total is a sum for all functions, and Max is the maximum value among functions. Value is a ratio of obfuscated and non-obfuscated program’s metrics. NSO stands for non-succeeding opcodes and unlimited stands for merge unlimited functions together. See B.2 for parameter explanation.

Obfuscation [settings]	AVL Tree		AES		QuickSort	
	Total	Max	Total	Max	Total	Max
VM [default]	7.57	11.42	16.60	41.40	3.55	9.90
VM [customSwitch, NSO]	7.35	11.33	16.24	41.20	3.40	9.80
VM [allocs]	8.93	13.50	15.02	37.00	3.22	8.30
ShuffleBBs [default]	1.00	1.00	1.00	1.00	1.00	1.00
Interleaving [default]	1.17	1.33	1.10	2.60	1.15	1.80
Interleaving [unlimited]	1.15	2.92	0.86	4.60	1.01	1.80
Interleaving [deadFns=0]	1.24	1.33	0.98	1.80	1.06	1.80

Table 5.2: Cyclomatic complexity changes (μ_2) for selected programs. Total is a sum for all functions, and Max is the maximum value among functions. Value is a ratio of obfuscated and non-obfuscated program’s metrics. NSO stands for non-succeeding opcodes and unlimited stands for merge unlimited functions together. See B.2 for parameter explanation.

5.1.3 Resilience

Resilience measures how difficult it is to create an automatic deobfuscator and how much execution time it will require. Collberg et al. [2] defined resilience on a five-point scale: *trivial*, *weak*, *strong*, *full* and *one-way*. Below, we will discuss resilience for each implemented obfuscation transformation.

5.1.3.1 Lightweight VM Pass

We see two possible ways of undoing this transformation. The first one is to manually/semi-automatically understand used opcodes and write disassembler. Since this pass operates on functions and use randomized opcodes for every obfuscated function, the creation of disassembler could be a quite difficult task, but the created disassembler should be very efficient and require minimum running time. The second way is to detect the VM interpreter automatically, all obfuscated BBs within a particular function, assignment of an array with opcodes to a shared variable for the VM interpreter and undo all operations. This would be quite tricky to do it in a general way, as every function is obfuscated randomly and some of BBs might not be obfuscated.

In [2] is the resilience of this pass estimated as *strong*. However, our VM works on the function level, so we consider its resilience to be *weak*.

5.1.3.2 String Encryption Pass

For undoing the String Encryption, there are also two ways. The first one includes detecting at least one decryption function, extracting it from a binary, running it to decrypt all obfuscated strings and then replacing all encrypted

strings with the decrypted ones. The second one is detecting the used decryption algorithm, decrypting all strings and replacing them.

Both ways have pros and cons, extracting decryption functions could be a bit tricky but also detecting the encryption algorithm is tricky, as we did several countermeasures to make this process more difficult. For instance, we use up to twelve decryption functions with inlined decryption code and masked AES tables.

Anyway, the main issue is that even strings may provide a lot of clues, undoing this pass would probably require much more effort than the possible result, and the reverse-engineer will probably give-up and focus more on reverse-engineering program's logic. Since this pass affects only BBs, we estimated its resilience to be *trivial*.

5.1.3.3 ShuffleBBs Pass

Undoing this pass probably would not make much sense, as it is not worth the effort and it could be tricky when used together with VM or Table Interpretation. We estimated the resilience of this pass as *weak*.

5.1.3.4 Improved Interleaving Pass

To undo this pass, reverse-engineer will have to detect the interleaved functions and extract the interleaved body's back to separate functions. In [2] is the resilience of this pass estimated as depending on used opaque predicates. Our implementation does not use opaque predicates, so we see the resilience as *weak*. However, its resilience can be increased by using this pass in combination with other passes such as virtualization.

5.1.4 Performance impact

In this subsection, we will evaluate the performance impact of our obfuscations. In [2] the performance impact is measured on a four-point scale: free, cheap, costly and dear. The testing environment and used algorithms were described at the beginning of this section. The input data were randomly generated and have not changed during testing. All programs were run ten-times, and the average was calculated.

To measure the performance impact, we recorded run times for the both non-obfuscated and obfuscated variants and also used them to calculate slow-down ratios between obfuscated and non-obfuscated variants.

We have decided not to measure the performance impact of String Encryption and ShuffleBBs using the selected algorithms. For the String Encryption, we have the same reason as in case of measuring its potency. However, we will measure the performance of String Encryption separately. ShuffleBBs does not have almost any performance impact, and it is completely pointless to measure it.

5.1.4.1 Lightweight VM Pass

We measured VM with (a) default settings, (b) custom switch and non-succeeding opcodes and (c) with allocation instructions.

In Figure 5.1 we can see the performance of elements insertion to AVL Tree. The both (a) and (b) provides slowdown around 3.5. (c) causes much higher slowdown, from 7 to 12. The slowdown of inserting 250 thousand elements is lower than in other near cases. If we take a look at execution times, the reason for the lower slowdown is that the execution of the non-obfuscated version was slower. Since all tests were run ten times, we can exclude that the difference is caused by the observational error. The more meaningful reason could be, for instance, caching.

In Figure 5.2 we can see the performance of AES Encryption. (b) leads the lowest slowdown, from 12 to 18 and the both (a) and (c) have similar slowdown from 20 to 27.

In Figure 5.3 we can see the performance of QuickSort. (a) is the fastest one with slowdown around 25. The both (b) and (c) have similar slowdown from 27 to 33. From these results, it seems that our implementation handles recursions poorly.

In the design part of this thesis, we estimated slowdown around 12. It is interesting that in the case of AVL the slowdown is much lower and in the rest of cases much higher. The most probable reason is in the backend and its transformation from LLVM IR to target architecture. We do not see any correlation between the input size and the slowdown, but the slowdown is much higher than in all other cases. Therefore, we estimated the performance impact as *costly*.

5.1.4.2 Improved Interleaving Pass

We measured interleaving with default settings and with settings to interleave unlimited functions together. The performance results are not very visible in the produced graphs due to the strong difference between the performance impact of virtualization and interleaving.

In case of elements insertion to AVL Tree (Figure 5.1), both variants of settings produce similar results, and the slowdown is from 0.95 to 1.6. In case of AES Encryption (Figure 5.2), the variant with default settings has better results with the slowdown from 0.8 to 1.1 and the second variant has the slowdown from 1.05 to 1.6. Finally, in case of QuickSort (Figure 5.3), the variant with default settings has better results with the slowdown from 1.25 to 1.3 and the second variant has the slowdown from 1.25 to 1.4.

We do not see any correlation between the input size and slowdown ratio; thus the performance impact is *free*.

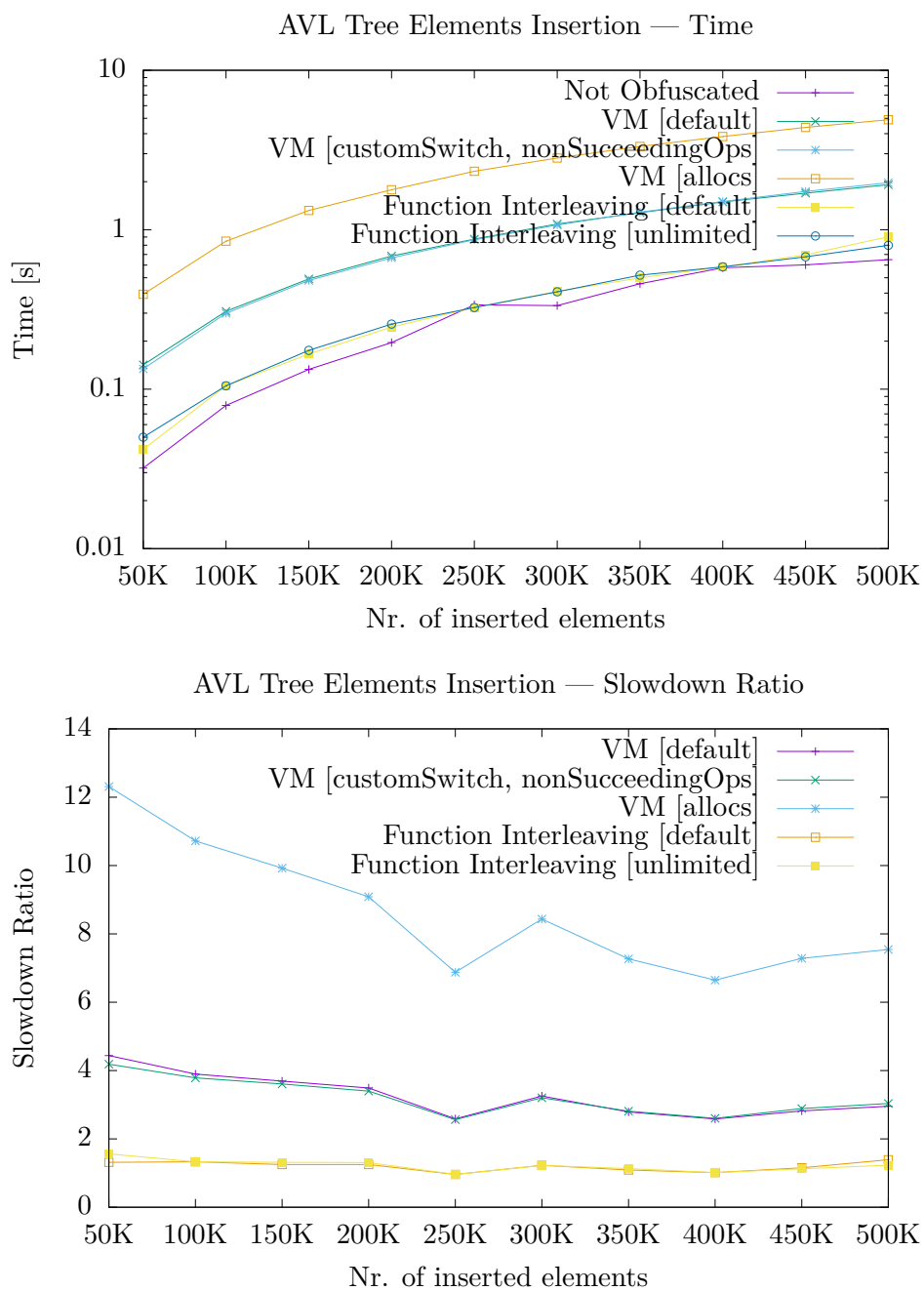


Figure 5.1: AVL Tree Elements Insertion performance

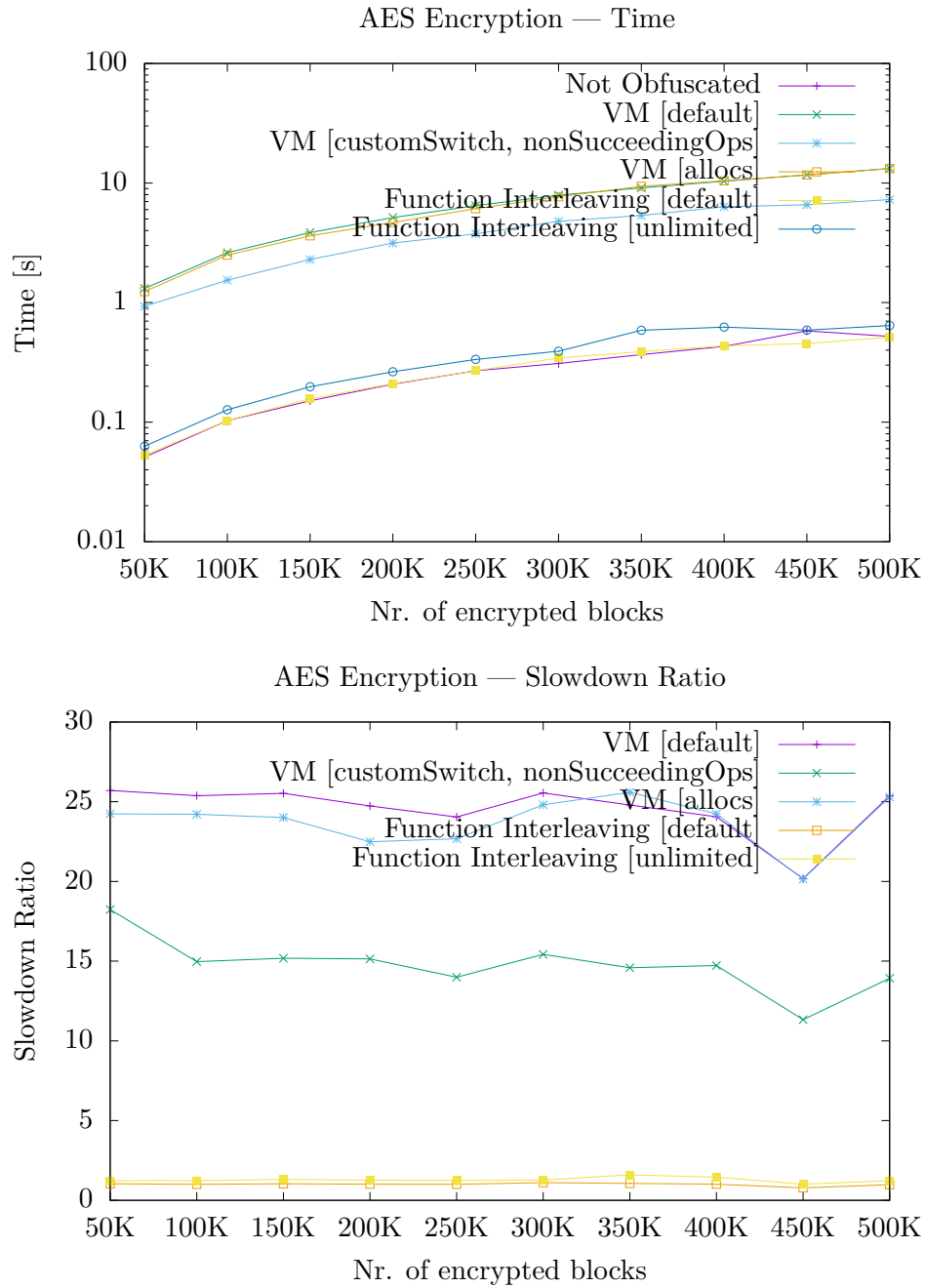


Figure 5.2: AES Encryption performance

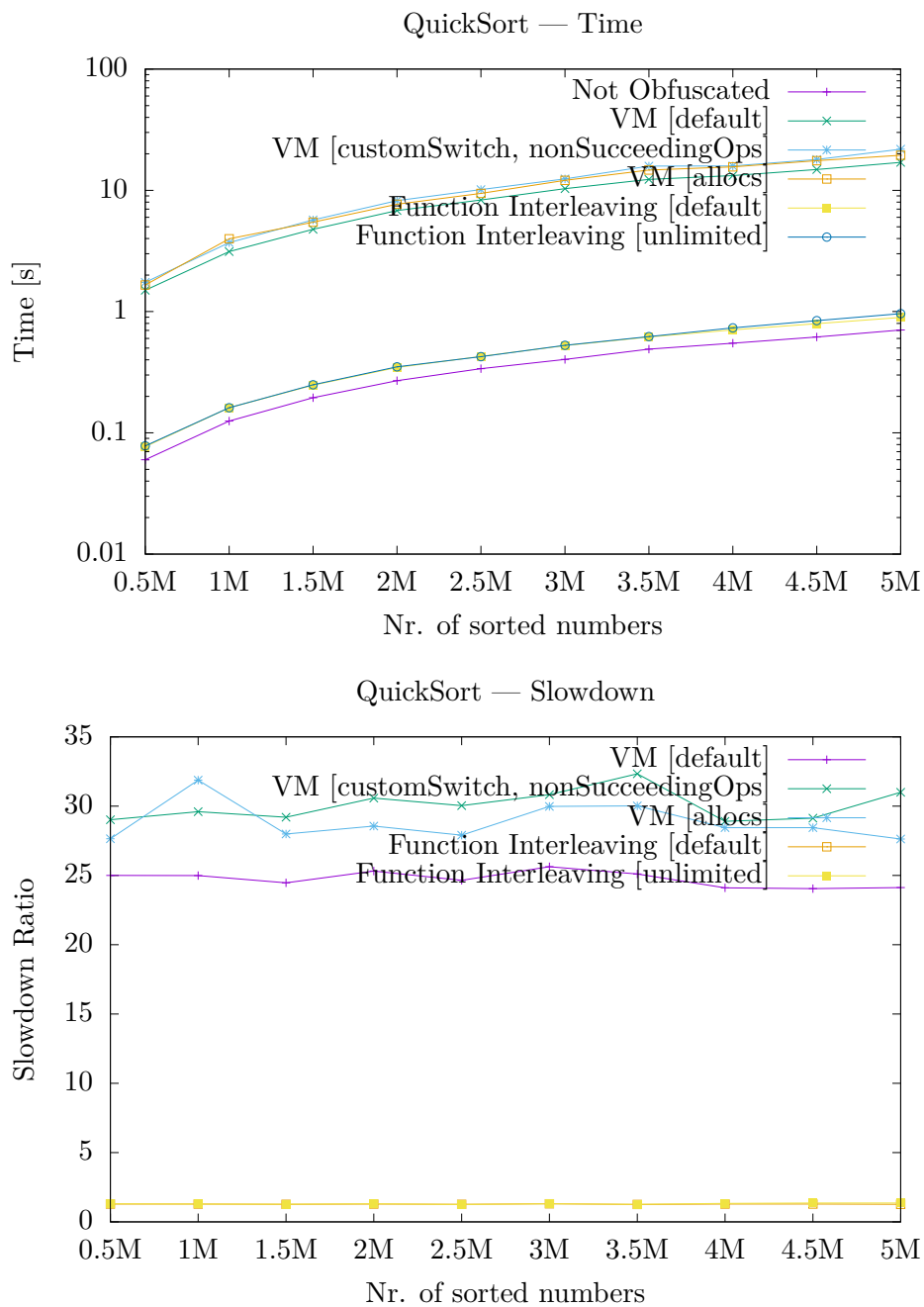


Figure 5.3: QuickSort performance

5.1.5 Summary

In this section, we evaluated potency, resilience and performance impact of created obfuscation transformations. In Table 5.3 you can see the summary.

Obfuscation	Potency	Resilience	Cost
Lightweight VM	strong	weak	costly
StringEncryption	—	trivial	free
ShuffleBBs	low	weak	free
Improved Interleaving	low	weak	free

Table 5.3: Evaluated obfuscation metrics. Performance impact of String Encryption was measured in section 5.2

In the case of Lightweight VM, the results differ from the evaluation in [2]. The differences emerge from different implementation, whereas they consider transforming a whole program to VM; our approach is per function. We also decided not to measure the potency of String Encryption Pass, as we think that any result would be misleading.

5.2 Evaluation of String Encryption Pass

In this section, we will measure the performance and size penalty of String Encryption. We decided to evaluate this pass, as measuring its parameters on regular programs would be unreasonable. In the design of String Encryption Pass, we set two goals, quick decryption and low increase of binary size. The evaluation will focus on these goals.

5.2.1 Test Programs

For the evaluation, we have generated programs in C language with random forty-characters-long regular and wide strings. Strings are printed to standard error output using *fprintf* or *fwprintf*. Length of wide characters is four bytes, and we made sure that decryption arrays to store decrypted strings will always be allocated on the stack using *allocainst*. The programs were compiled using clang without any optimizations to avoid string concatenation.

5.2.2 Decryption Performance

In Figure 5.4 and Figure 5.5 we can see the performance of decryption process, either in absolute values or a ratio of encrypted strings to plain ones. In the case of regular strings, the slowdown is around two. More mystical is the case of wide strings, where the slowdown goes from 2.5 to 4.5. The increase of

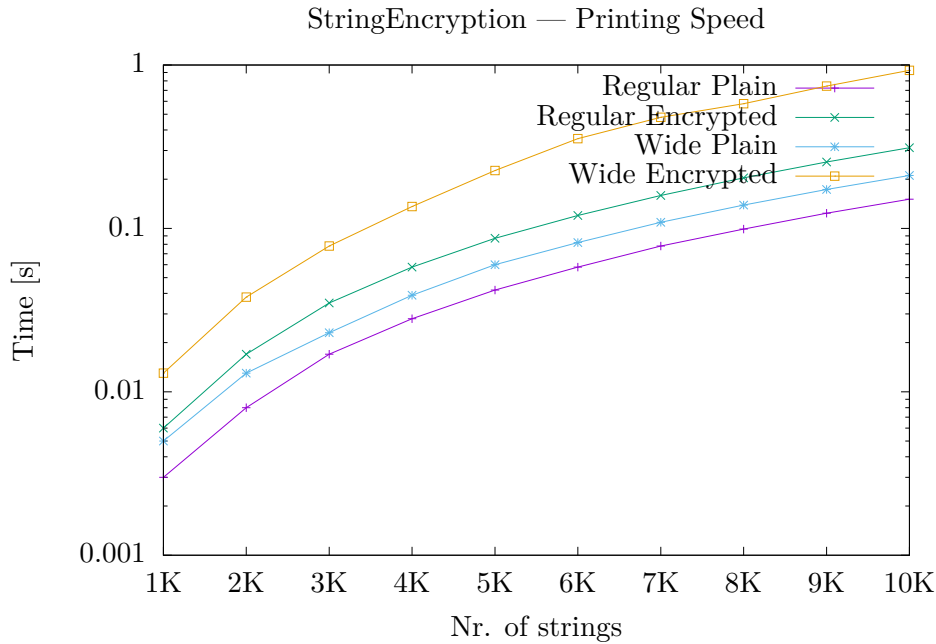


Figure 5.4: Decryption performance — Printing Speed

slowdown is probably caused by the allocation of bigger arrays on the stack to store decrypted strings.

Even the slowdown up to 4.5 may seem huge; we have to realize that the tested program contains only strings which is not a case of regular programs, where logic takes a much bigger portion of binary size than strings. If we consider that the decryption of ten thousand forty-characters-long wide strings takes less than one second, we can say that the decryption is quick and one of the goals is fulfilled. String encryption should not correlate with the algorithm's input size, so we evaluated the performance impact as *free*.

5.2.3 Size of Encrypted Strings

In Figure 5.6 we can see the increase of binary size with encrypted strings. It is from 2.4 to 2.6 for regular strings and 1.5 for wide strings. The reasons why the encryption of regular leads to bigger binary are following. Rounding of string length to multiplications of AES block size makes a higher relative increase in case of a binary with regular strings. Decryption keys and decryption logic also take a higher relative portion of binary size in case of a binary with regular strings.

The binary size could be smaller by using shared keys at the expense of decreased resilience. We think that the increase in size is reasonable and the goal of keeping the increase as low as possible is achieved.

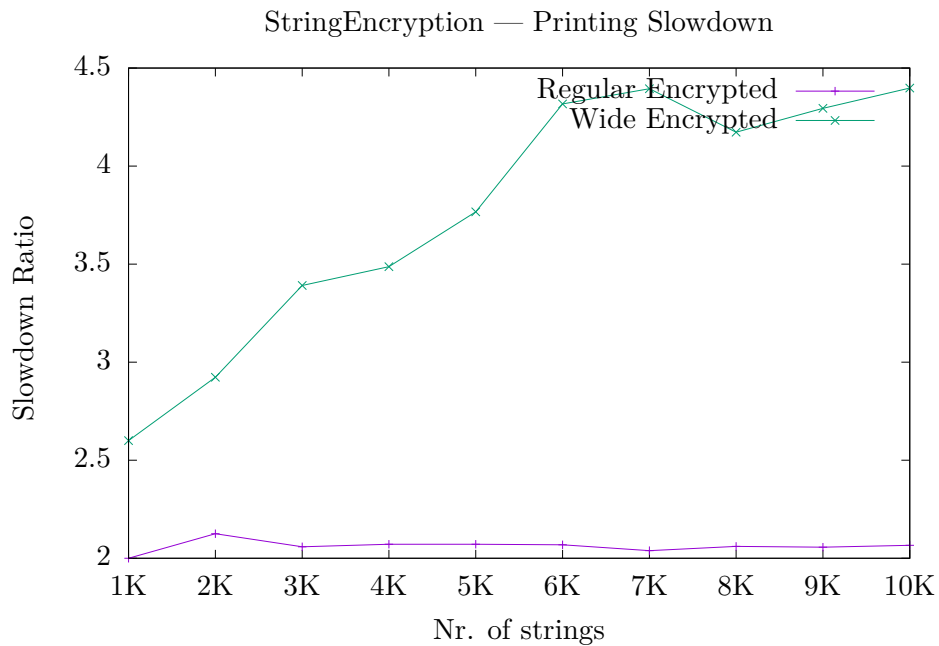


Figure 5.5: Decryption performance — Printing Slowdown

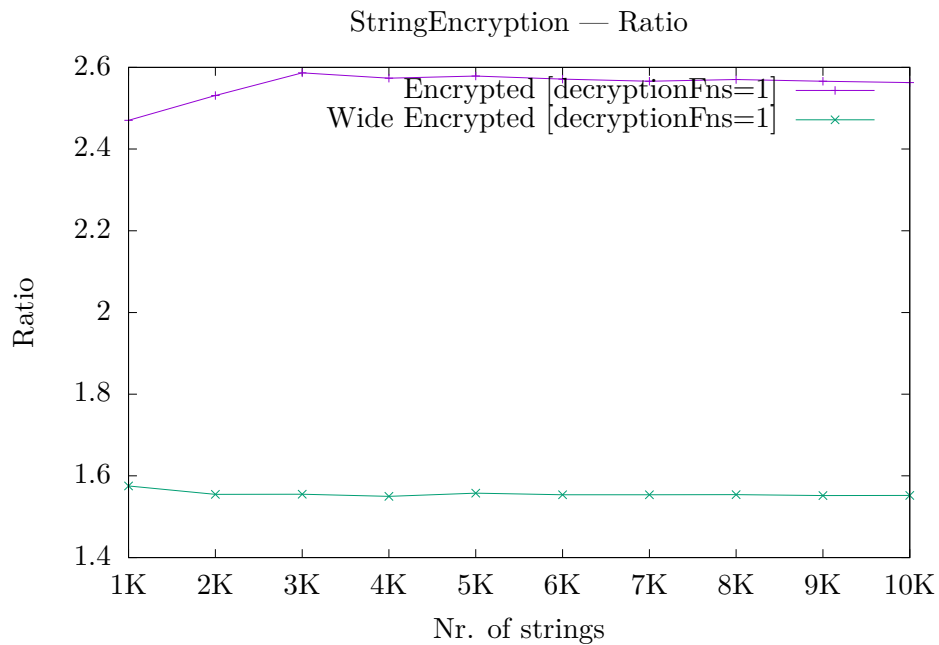


Figure 5.6: Comparison of binary sizes. The ratio of binary sizes with encrypted strings to binary sizes with plain strings.

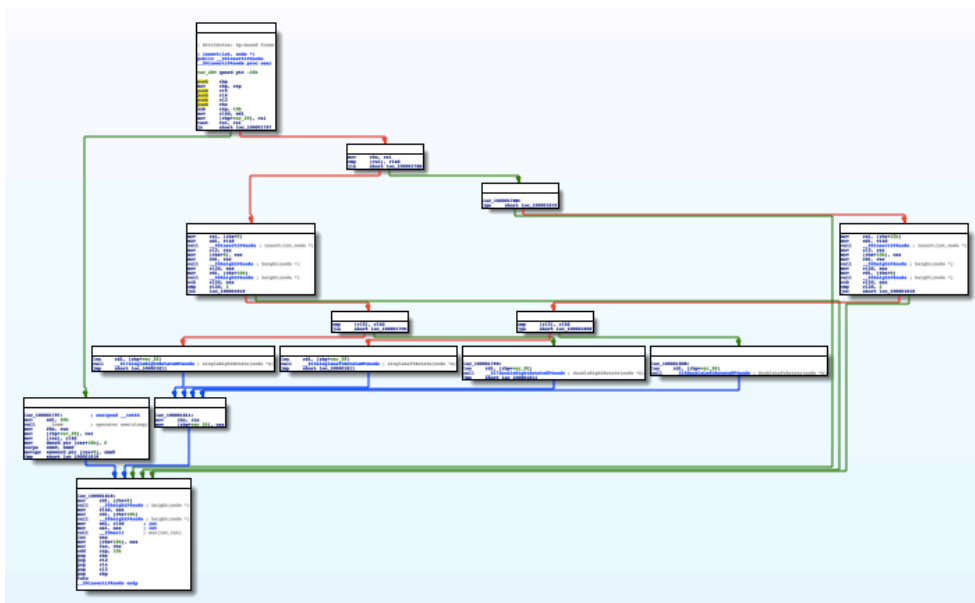


Figure 5.7: Visualization of AVL’s insert function in IDA. Non-obfuscated version.

5.3 Quality of Lightweight VM

To subjectively assess the quality of VM obfuscation, we take a look at the visualization of AVL Tree’s insert function in IDA. In Figure 5.7 we can see the non-obfuscated version, which looks pretty simple, in Figure 5.8 we can see the VM obfuscated version with default options. Finally, in Figure 5.9 we can see VM obfuscated version with custom switch and non-succeeding opcodes. At the first look, we can say that VM adds a lot of confusion and destroys the function’s CFG even for a simple function. If we take a deep look at Figure 5.9, we can see a custom implemented switch, based on a tree structure.

Another disadvantage in addition to the huge performance impact of VM obfuscation is that the obfuscated function is, as you can see in the visualizations, obvious and it can motivate reverse-engineer to prioritize these functions as they may contain something important.

5.4 Comparison with Existing Tools

5.4.1 Commercial Tools

VMProtect and Themida support virtualization, even we do not have exact implementation details, we can guess that their virtualization is probably much more optimized and provides better performance. Otherwise, they would not

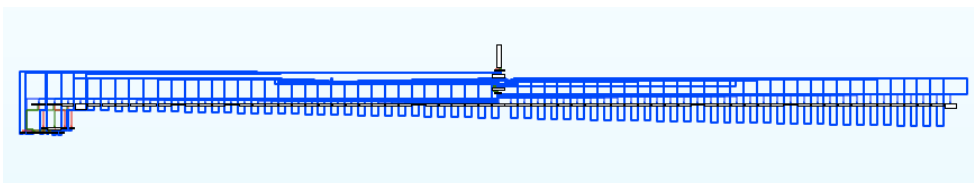


Figure 5.8: Visualization of AVL's insert function in IDA. VM-obfuscated with default options.

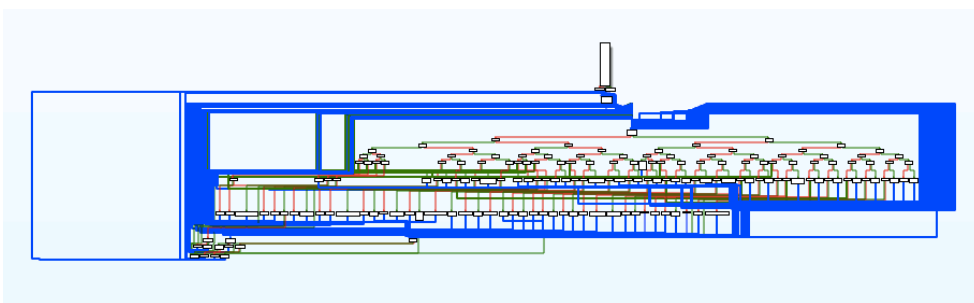


Figure 5.9: Visualization of AVL's insert function in IDA. VM-obfuscated with custom switch and non-succeeding opcodes.

be able to sell their tool massively. On the other hand, these tools support only PE while our tool is source language and target architecture agnostic.

PELock offers changes of the code execution flow, mutation of original instructions, hide of calls and insertion of dead code. These obfuscation transformations do not seem to be as good as virtualization. CXX-OBFUS offers only layout obfuscations, which is far beyond virtualization.

5.4.2 Open-source Tools

Tigress C Obfuscator supports virtualization, more advanced and efficient than one implemented in this thesis. Additionally, it also supports transforming functions to generate their machine code at runtime or continuously modify their machine code. This tool is far beyond the one created in this thesis. Unfortunately, it supports only C language and requires a lot of fine-tuning.

Swift Shield and iOS Class Guard provide only layout obfuscations. Obfuscator-LLVM does not provide as strong obfuscation transformations as we implemented in this thesis, or were implemented in [15, 17].

Petráček's [15] and Šíma's [17] obfuscators offer Table Interpretation, a very significant obfuscation transformation. Obfuscator-LLVM provides Table Interpretation too but in a simple form. Virtualization implemented in this thesis creates a better confusion but at the expense of a huge performance

penalty while Table Interpretation is a great trade-off between quality and performance.

5.5 Summary

In this section, we compared our tool to existing obfuscation tools. VMProtect and Tigress C Obfuscator offer the same type of obfuscation transformation as our tool, implemented with higher efficiency. Also, these tools have a lot of users, have been widely tested and probably are much more stable than our tool. Anyway, they are limited to specific source languages and target architectures.

Petráček's [15] and Šíma's [17] obfuscators offer Table Interpretation which can be used widely, whereas our Virtualization due to performance penalty should be used only on functions that require a higher level of protection.

Conclusion

The goals of this thesis were: a) to explore the taxonomy of obfuscation transformations, b) to get acquainted with the LLVM Framework, c) to design and implement an LLVM-based obfuscator, d) to implement the transformations in such a way that they support parameters and pragmas to control the obfuscation process, e) to analyze the potency and resilience of each implemented transformation, f) to compare the results with existing tools, and g) combine selected transformations from [15, 17] together with the transformations created in this thesis.

At the beginning of this thesis, we formally introduced obfuscation transformations, mentioned concrete examples and evaluation metrics. We put forward the LLVM Framework, described its philosophy and essential parts. Then we designed and implemented the following obfuscation transformations: Lightweight VM, String Encryption, ShuffleBBs, and Improved Function Interleaving.

The main achievement of this thesis is that it proves the possibility of using the LLVM Framework to perform virtualization-based obfuscations. Virtualization is one of the state-of-the-art transformations available nowadays, and also the most advanced transformation we implemented.

Nevertheless, we can find available tools offering more efficient and stable virtualization or even more potent transformations, such as VMProtect, Themida and Tigress C Obfuscator. These tools have limitations as well. Either they are limited to a particular platform and source language, or they require complex configuration.

Our virtualization is not perfect and has its limits. Its performance penalty is volatile and highly dependent on a specific implementation of an algorithm. Furthermore, it also does not handle recursions efficiently. For instance, during the evaluation, we had to increase stack size to be able to sort five million numbers using the QuickSort algorithm.

In the evaluation, we demonstrated the power of virtualization and efficiency of string encryption that decrypts ten thousand wide strings under one second. We also found out that some transformations have different potency and resilience than the estimates in [2]. This is due to differences in their design and implementation. All implemented transformations were evaluated as having constant slowdown, unrelated to the input size.

Our obfuscation tool includes transformations implemented in [15, 17] and for all transformations, we implemented support for CLI parameters and pragmas. The Appendices section includes information about usage and a list of all available transformations, parameters, and pragmas that allow fine-grained control of the obfuscation process.

6.1 Future Work

Our implementation of lightweight virtualization opens an unlimited space for enhancement and further research. The main areas of potential improvements are performance and quality. Any enhancement in these areas, discussed below, is a possible continuation of this thesis.

One of the implementation downsides leading to decreased performance is that every occurrence of instruction is considered to be unique and added separately to the VM Interpreter. For instance, any two identical instructions get assigned two different opcodes. While this increases potency and resilience, it also increases the performance penalty and binary size. The transformation was implemented this way due to the architecture of LLVM IR which does not allow dynamic assignment of operands or virtual registers. This issue could be solved by implementing support for dynamic assignment in LLVM IR. Improving performance on recursions is also an opportunity for significant performance improvement.

To increase potency or resilience, it is a possibility to move the VM Interpreter outside of the obfuscated functions and make it shared between all obfuscated functions. This would add more confusion as instructions in the VM Interpreter would be originating from various functions.

As a part of this thesis, we implemented virtualization that obfuscates functions on the basic block level. This could be improved upon by obfuscating the whole function, so its former basic blocks (except the entry one) will be completely removed. This would make the obfuscation more resilient at the expense of limiting the granularity of configuration.

Bibliography

- [1] Barak, B.; Goldreich, O.; et al. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology — CRYPTO 2001*, Springer Berlin Heidelberg, 2001, ISBN 978-3-540-44647-7, pp. 1–18.
- [2] Collberg, C.; Thomborson, C.; et al. A Taxonomy of Obfuscating Transformations. Technical report 148, Department of Computer Sciences, The University of Auckland, 1997.
- [3] 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, ISBN 0-89791-979-3, pp. 184–196.
- [4] Cytron, R.; Ferrante, J.; et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, volume 13, no. 4, 1991: pp. 451–490, ISSN 0164-0925.
- [5] Hamann1, M.; Krausel, M.; et al. A Note on Stream Ciphers that Continuously Use the IV. Available from: <https://eprint.iacr.org/2017/1172.pdf>
- [6] Horwitz, S. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, volume 19, 1996.
- [7] Kuang, K.; Tang, Z.; et al. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security*, volume 74, 2018: pp. 202–220.
- [8] Lattner, C. *The Architecture of Open Source Applications: LLVM*. lulu.com, 2011, ISBN 978-1-257-63801-7.

- [9] Lattner, C.; Adve, V. The LLVM Instruction Set and Compilation Strategy. Available from: <https://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf>
- [10] LLVM Team. LLVM Language Reference. Accessed on 19.01.2019. Available from: <https://llvm.org/docs/LangRef.html>
- [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] Majumdar, A.; Thomborson, C.; et al. A Survey of Control-Flow Obfuscations. In *Information Systems Security*, Springer Berlin Heidelberg, 2006, ISBN 978-3-540-68963-8, pp. 353–356.
- [13] Muchnick, S. *Advanced Compiler Design and Implementation*, chapter Approaches to control-flow analysis. Morgan Kaufmann, 1997, ISBN 1558603204, pp. 172–177.
- [14] Namjoshi, K. S. Witnessing An SSA Transformation. Available from: <https://kedar-namjoshi.github.io/papers/Namjoshi-VeriSure-CAV-2014.pdf>
- [15] Petráček, M. *LLVM Obfuscator*. Master’s thesis, Czech Technical University, Faculty of Information Technology, 2018.
- [16] Rijmen, V.; Bosselaers, A.; et al. Optimised ANSI C code for the Rijndael cipher (now AES). Accessed on 13.01.2019. Available from: <https://fastcrypto.org/front/misc/rijndael-alg-fst.c>
- [17] Šíma, R. *Obfuskátor nad platformou LLVM*. Master’s thesis, Czech Technical University, Faculty of Information Technology, 2018.

Acronyms

API Application Programming Interface

BB Basic Block

CFG Control Flow Graph

LLVM Low Level Virtual Machine

SSA Single Static Assignment

PE Portable Executable

Usage

In this chapter, we will describe how to use the created obfuscation tool.

B.1 Getting Started

To use the obfuscator, we need to compile the LLVM Framework with the obfuscator by following the steps below:

1. Download LLVM 7.0.1 sources.
2. Copy the directory with obfuscator's sources located on the enclosed drive in "SRC/Obfuscator" to "lib/Transforms/Obfuscator".
3. Append a file located in "lib/Transforms/CMakeLists.txt" with a line "add_subdirectory(Obfuscator)".
4. To finish the compilation of LLVM follow this article:
<https://llvm.org/docs/CMake.html>.

Once the build is ready, the obfuscator should be compiled as a library called *Obfuscator.dylib* (OS X) or *Obfuscator.so* (GNU/Linux). Note: The process is more complicated on Windows Platform¹⁸. To use *opt* with the library, we need to specify the library by using *load* parameter.

An example of usage:

```
$ opt -load=Obfuscator.dylib -nameOfPass -parameter -S  
  ↪ source.ll -o obfuscated.ll
```

To list all available obfuscation transformations:

¹⁸<https://lists.llvm.org/pipermail/llvm-dev/2015-November/091960.html>

B. USAGE

```
$ opt -load=Obfuscator.dylib -help-list | grep "\-obf" |  
  ↪ grep "Pass"
```

To list all available parameters of obfuscation transformations:

```
$ opt -load=Obfuscator.dylib -help-list | grep "\-obf" |  
  ↪ grep -v "Pass"
```

B.2 Available Passes & Parameters

In this section, we will list the all available passes and accepted CLI parameters or pragmas to gain control of obfuscation transformations. CLI Parameters override default values and pragmas override parameters for a specific function or variable. Parameters are simply appended to the *opt* command and pragmas are used as described in 3.3.1.

List of available obfuscation transformations and their parameters:

- Pragma available in all passes:
 - obf-skip: A function will not be obfuscated, or string will not be encrypted.
- Lightweight VM Pass (-obf-vm)
 - CLI Parameters:
 - obf-vm-allocs: *Allocainsts* will be obfuscated. May cause issues. [default=false]
 - obf-vm-custom-switch: Use a custom switch implementation. Usually slower than *switchinst*. [default=false]
 - obf-vm-only-marked-functions: Obfuscate only marked functions. [default=false]
 - obf-vm-succeeding-opcodes: Use random succeeding opcodes. Using random non-succeeding opcodes may cause issues when used without a custom switch implementation. [default=true]
 - Pragmas applicable to functions:
 - obf-vm / obf-vm-skip: A function will (will not) be converted to the VM representation.
 - obf-vm-allocs / obf-vm-no-allocs
 - obf-vm-succeeding-opcodes / obf-vm-non-succeeding-opcodes
 - obf-vm-max-bbs=<unsigned>: The maximum number of function's basic blocks converted to the VM representation. [default=0, 0 means unlimited]

Note: Lightweight VM Pass should always be applied as the last transformation.

- String Encryption VM Pass (-obf-string-encryption)
 - CLI Parameters:
 - obf-string-encryption-decryption-fns=<uint>: The number of decryption functions [min=1, max=12, default=2] with inlined decryption code. Each function will use dedicated *XORed* tables (6x1KB). One of these functions will be randomly selected for each decryption.
 - Pragas applicable to variables:
 - obf-string-encryption-skip: A string or array of strings will not be encrypted.
- ShuffleBBs Pass (-obf-shuffle-bbs)
 - Pragas applicable to functions:
 - obf-shuffle-bbs-skip: A function will be skipped from obfuscation.
- Function Interleaving Pass (-obf-fn-interleaving)
 - CLI Parameters:
 - obf-fn-interleaving-max-dead-fns=<uint>: The maximum number of *dead functions* added to interleaved functions. [default=2]
 - obf-fn-interleaving-max-fns=<uint>: The maximum number of interleaved functions. [default=10, 0 means unlimited]
 - obf-fn-interleaving-max-interleaved-fns-together=<uint>: The maximum number of functions merged together into a new function. [default=3, 0 means unlimited]
 - Pragas applicable to functions:
 - obf-fn-interleaving-skip: A function will be skipped from obfuscation.

Note: If we set *-obf-fn-interleaving-max-dead-fns=0* and *-obf-fn-interleaving-max-interleaved-fns-together=2*, the pass will fully fall back to implementation from [15] and use opaque predicates.

- Table Interpretation Pass (-obf-table-interpretation)
 - Pragas applicable to functions:
 - obf-table-interpretation-skip: A function will be skipped from obfuscation.

- Bogus Control Flow Pass (-obf-bogus-flow)
 - Pragas applicable to functions:
 - obf-bogus-flow-skip: A function will be skipped from obfuscation.
- Inlining Pass (-obf-inlining)
 - Pragas applicable to functions:
 - obf-inlining-skip: A function will be skipped from obfuscation.
- Opaque Predicates Pass (-obf-opaque-predicates)
 - Pragas applicable to functions:
 - obf-opaque-predicates-skip: A function will be skipped from obfuscation.
- Outlining Pass (-obf-outlining)
 - Pragas applicable to functions:
 - obf-outlining-skip: A function will be skipped from obfuscation.
- Split Blocks Pass (-obf-split-blocks)
 - Pragas applicable to functions:
 - obf-split-blocks-skip: A function will be skipped from obfuscation.
- Dead Code Pass (-obf-dead-code)
 - CLI Parameters:
 - obf-dead-code-irr-ratio=<uint>: The ratio between dead and irrelevant basic blocks (min=1, max=100). [default 50]
 - obf-dead-code-new-ratio=<uint>: The ratio of newly added and all basic blocks (min=1, max=100). [default 50]
 - Pragas applicable to functions:
 - obf-dead-code-skip: A function will be skipped from obfuscation.

B.3 String Encryption Pass

String Encryption Pass is different from the rest because after using it, we need to link the result with the decryption functions. The file containing decryption functions is located on the enclosed drive in “SRC/rijndael-*alg-fst.c*”.

See an example of string encryption process below:

```
$ clang -emit-llvm -S source.c
$ opt -load=Obfuscator.dylib -obf-string-encryption -obf
  ↪ -string-encryption-decryption-fns=12 -S source.ll
  ↪ -o encrypted.ll
$ clang -emit-llvm -S rijndael-alg-fst.c
$ llvm-link -only-needed -S rijndael-alg-fst.ll
  ↪ encrypted.ll -o linked.ll
$ llc linked.ll
$ clang linked.s -o binary
```

B.4 Debug Mode

B.4.1 Assertions with Lightweight VM Pass

If we would like to use LLVM in debug mode with enabled assertions, we need to make a tiny adjustment of LLVM’s source code by following the steps below. Otherwise, Lightweight VM Pass would not work correctly.

1. Open *lib/IR/Verifier.cpp*.
2. Find and comment out the assert containing “Instruction does not dominate all uses!”.


B.4.2 Debug Messages


In order to see more detailed messages about obfuscation process, LLVM must be compiled in the *debug* mode. We can see debug messages by appending *-debug-only=nameOfPass* to the *opt* command.

An example of running Lightweight VM Pass with debug messages:

```
$ opt -load=Obfuscator.dylib -obf-vm -debug-only=obf-vm
  ↪ -S source.ll -o obfuscated.ll
```



Contents of enclosed CD


 DP_Turcan_Lukas_2019

 `readme.txt` — the file with CD contents description

 SRC — the directory of source codes

 EXAMPLES — the directory of example programs

 TEXT — the thesis text directory

 `thesis.pdf` — the thesis text in PDF format