Masaryk University
Faculty of Informatics

Modern obfuscation techniques

Master's Thesis

Roman Oravec

Brno, Fall 2021

Masaryk University
Faculty of Informatics

Modern obfuscation techniques

Master's Thesis

Roman Oravec

Brno, Fall 2021

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Roman Oravec

Advisor: Mgr. et Mgr. Jan Krhovják, Ph.D.

Acknowledgements

I want to thank my advisor, Mgr. et Mgr. Jan Krhovják, Ph.D., for giving me the freedom in exploring a field I am interested in, as well as his guidance and advice. I would also like to express my gratitude to my close ones for supporting me throughout the process of writing this thesis.

Abstract

In the context of cybersecurity, obfuscation is a method of manipulating a computer program with the intention to obscure its inner workings. Various obfuscation techniques have found their use as a means to protect intellectual property and to prevent code tampering, as well as to achieve malicious purposes, such as creating malware that can circumvent detection mechanisms. In this thesis, we examine multiple commonly used obfuscating techniques and freely available tools that implement them. We also discuss the possibilities of the LLVM Pass Framework for obfuscating programs during compilation. In the practical part, we have implemented several obfuscating transformations, leveraging the LLVM Pass Framework. The quality of the implemented transformations has been tested and evaluated with respect to selected metrics. Lastly, we have examined and tested the possibility of using our implementation to obfuscate programs written in C/C++ for Android OS.

Keywords

obfuscation, LLVM, compilers, security, Android, C++

Contents

1 Introduction

When distributing proprietary software, the authors usually face the challenge of providing its functionality to the users without disclosing too many details about the implementation, while they also want to prevent any unauthorized attempts to modify their product. They want to protect their intellectual property, forbid the users from illegally distributing their products, and prevent the competition from stealing their ideas, namely the algorithms.

One of the solutions to this issue is to execute the critical parts of the software on the remote server, and sell it to the users as a service. A similar approach is to use a digital rights management (DRM) solution that remotely verifies that the software running on the user's machine is legitimate. However, both of these approaches require internet connection, which is deemed unnecessary from the users' point of view in case of software that could be used offline without this measure of protection (e.g., single-player games, graphics software). It might negatively affect the user experience in case of failure on the server side, as in the recent case of Denuvo, when the unavailability of this DRM service prevented many users from playing several video game titles [1].

A different solution for this problem is obfuscation. By obfuscating a program, the developers aim to modify it in a way that preserves its functionality, but makes it harder to understand, or reverse engineer, either manually or with the help of tools for automated analysis. As shown by Barak et al. [2], it is not possible to create a perfect obfuscation and with enough time and resources, any program can be deobfuscated. However, the goal of obfuscation is to discourage a reverse engineer from analyzing or tampering the code, by transforming a program in a way that any such attempts would be infeasible.

Obfuscation is also widely used by malicious actors, who aim to produce malware that can bypass automated analysis tools, e.g. an antivirus software. Furthermore, their goal is to confuse a reverse engineer so that it would take more time and resources to analyze the malware and develop countermeasures. Both malicious and benign parties keep trying to come up with more advanced obfuscating transformations as well as new reverse engineering techniques, and the result is a form of a cat-and-mouse game that accelerates the research in this area.

In a pivotal paper published in 1997 [3], Collberg et al. proposed a variety of obfuscating transformations, together with recommendations on how to categorize and evaluate them. Many techniques used in the present are based on the ones described in this paper. Most of them can be sorted into two categories – control obfuscation and data obfuscation. Techniques in the first category manipulate the flow of control in the program, either by breaking up or merging computations, randomizing their order, or adding redundant code. The latter category includes transformations that obscure data such as character strings and constants, for example by changing their encoding or making the program compute them during runtime, and thus not storing them statically as a part of the executable file.

Obfuscation can be considered as a part of the compilation process. The LLVM Framework [4] provides a convenient way to manipulate a program in an automated manner with transformation passes. A program compiled with LLVM is first transformed into an intermediate representation, then the transformations can be applied, and finally, machine code is generated. The original purpose of the transformation passes is to perform optimization to make the resulting program more efficient. However, it also allows developers to automatically obfuscate a program. As a part of this thesis, we aim to examine the possibilities of this framework and use it to implement obfuscating transformations. Furthermore, we are going to evaluate them in terms of resilience against reverse engineering attempts, and also their impact on the program performance. We are also going to explore the possibility of using the implementation to obfuscate executables compiled for devices with ARM processors running Android OS.

The structure of the thesis is as follows: chapter 2 introduces various obfuscating transformations and a way to evaluate their qualities, Chapter 3 describes the LLVM Framework with a focus on its components that are useful for implementing obfuscation, in Chapter 4, freely available tools for obfuscation are being discussed, Chapters 5, 6 and 7 deal with the design, implementation and evaluation of the obfuscating transformations, and 8 describes the possibilities of using obfuscation based on the LLVM Framework on Android OS, together with a practical example.

2 Obfuscation

This chapter describes some of the obfuscation techniques from a general point of view, as those principles can be applied to obfuscate a program on the source code, intermediate representation, or machine code level. The subsequent chapters will deal with implementation details for applying those methods to a program represented in the LLVM Intermediate Representation.

2.1 Obfuscating Transformations

Collberg et. al [3] formally define an *obfuscating transformation* as follows:

Definition 2.1.1 (Obfuscating transformation)**.**
Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{T} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{T} P'$ to be a legal obfuscating transformation, the following conditions must hold:

- If P fails to 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.

This definition states that applying the transformations should preserve the input-output behavior of the program, but allows the obfuscated version of the program to produce some additional behavior, for example performing calls to the operating system or creating new files.

The transformations may, and usually do, increase the computational complexity and memory requirements of the program.

2.1.1 Opaque predicates

Opaque predicates are common but effective constructs used for obfuscation. Constructing an opaque predicate consists of transforming a simple boolean expression into a complex one.

Definition 2.1.2 (Opaque predicate)**.** A predicate P is opaque at point p in a program, if its outcome is known at obfuscation time. We write P_p^F (P_p^T) if P always evaluates to `False` (`True`) at p, and $P_p^?$ if P may sometimes evaluate to `True` and sometimes to `False` [5].

Based on the possible values of an opaque predicate, we can distinguish two types:

Invariant opaque predicates. The value of the predicate is fixed, i.e. the obfuscator knows whether it evaluates to `True` or `False`, but its outcome is hard to deduce for the reverse engineer performing static analysis.

Two-way opaque predicates. This type of predicates can evaluate either to `True` or `False` for all possible inputs. Collberg et al. [3] suggested using two-way opaque predicates as a branching point to two functionally identical branches, which can be created by cloning a single branch and applying different obfuscations on both of them. This makes the static analysis of the program harder while preserving the original functionality.

Apart from types, opaque predicates can be categorized according to the methods of their construction:

Arithmetic-based opaque predicates. They are constructed using hard-to-solve mathematical formulas and used to hide invariant properties of the predicate. The obfuscator uses a mathematical identity that always evaluates to the same boolean value. In Collberg's taxonomy [3], this type of predicate falls into the *trivial* category, because the deobfuscator can deduce its value by performing a static analysis restricted to a single basic block[1] of a control flow graph.

This kind of predicates is used in the *Obfuscator-LLVM* in the *Bogus Control Flow* pass, which adds a new basic block to the Control Flow Graph, but the branch leading to this new block is never taken.

1. Basic block is a sequence of instructions executed one-after-the-other with no branching[6]

Mixed Boolean-Arithmetic opaque predicates. This method is based on a technique described by Zhou et al. [7]. MBA uses linear identities involving Boolean and arithmetic operations. MBA is further discussed in Section 2.1.5.

Alias-based opaque predicates. This is one of the resilient methods proposed by Collberg et al. [5] based on aliasing, i.e. state of a program where a particular memory location is referenced to by multiple pointers. When there is a possibility of aliasing, the static analysis might be much harder. In some cases, static alias analysis might be even undecidable [8].

Opaque predicates designed by Collberg et al. are constructed using two linked lists and two global pointers pointing into them. The opaque predicate then checks whether two pointers are aliasing, which always results in `False` since the two global pointers are constructed to always refer to nodes within different lists.

Environment-based opaque predicates. This method uses system calls or library calls which guarantee that the result of a predicate using such calls will always be either `True` or `False`.

An example of such opaque predicate is calling the `strcpy`[2] function from the C standard library and comparing the returned value (pointer to the destination array) with the first argument supplied to the function.

Bi-opaque predicates. This method of constructing opaque predicates has been recently introduced by Xu et al. [9]. The advantage of bi-opaque predicates is that they are resilient against symbolic execution engines, such as Triton[3], angr[4] and BAP[5]. The tool[6] developed by Xu et al. is built on *Obfuscator-LLVM*, but replaces the trivial arithmetic-based opaque predicates with more resilient *symbolic* opaque predicates, which exploit various weaknesses of symbolic execution. The *bi-opaque* property means that these predicates can introduce *false positives* to the analysis, which may make the reverse engineer falsely recognize legitimate predicates as opaque predicates.

2. https://www.cplusplus.com/reference/cstring/strcpy/
3. https://triton.quarkslab.com/
4. https://angr.io/
5. https://github.com/BinaryAnalysisPlatform/bap
6. https://github.com/zzrcxb/fusor

Opaque predicates can be used to enhance other obfuscation methods, for example, the dead code insertion, described in 2.1.4.

2.1.2 Instructions Substitution

Instructions substitution is one of the most simple obfuscation techniques. The principle of this method is to replace instructions containing binary arithmetic operations, such as addition and subtraction, and binary boolean operations, such as logical AND or XOR, with more complicated sequences of code, which yield the same result.

Despite its simplicity, this technique is still used in current obfuscation tools. For example, *Obfuscator-LLVM* uses simple substitutions using expressions composed exclusively of arithmetic operations to substitute addition and subtraction, and expressions composed of boolean operations to substitute boolean XOR, AND, and OR. Below are some of the substitutions implemented in *Obfuscator-LLVM*:

$$\begin{aligned}
&x + y \rightarrow -(-x + (-y)) \\
&x - y \rightarrow x + (-y) \\
&x \lor y \rightarrow (x \land y) \lor (x \oplus y) \\
&x \oplus y \rightarrow (\neg x \land y) \lor (x \land \neg y)
\end{aligned}$$

In contrast, *Tigress* implements obfuscations that transform standard binary operators with more complex expressions in MBA (2.1.5) form. A list of such identities can be found in [10], where the authors suggest using them to build efficient low-level operations for manipulating bit strings and numbers. Zhou et al. described a method to create and use such identities in [7], while also showing that there is an unlimited number of them.

This technique increases code diversification and it can be further improved by randomly choosing from several identities which can be used to substitute an instruction.

It is also important to note that the compiler can optimize out this kind of transformation, therefore it should not be used for source-level obfuscation. It also implies that it should be run after the optimization passes if we are obfuscating at the intermediate representation level.

A reverse engineer could use an optimizer on the obfuscated binary file to get rid of this transformation and reduce the complexity of the code for easier analysis. However, increased diversification of the code can still be useful, for example, to bypass an antivirus engine that is performing a static analysis of a program based on its signature, e.g., looking for known malicious patterns and sequences of instructions.

A rather bizarre example of this obfuscation technique is *The M/o/Vfuscator*[7], created by Chris Domas. This is a tool inspired by the idea, that the assembly instruction `mov` is *Turing-complete*. It is able to compile a program written in C language into assembly code composed exclusively of `mov` instructions. The resulting Control-flow graph looks like a straight line and it would be probably very hard and time-consuming to reverse engineer. However, this obfuscator is rather a proof of concept than a practical tool, due to its impact on the performance of the compiled program.

There is an even more extreme obfuscator worth mentioning, called `trapcc`[8]. It produces programs that are able to run without executing a single instruction of the CPU, leveraging Intel's Memory Management Unit (MMU) fault handling mechanism. However, this is probably not considered an instructions substitution transformation in the true sense.

2.1.3 Garbage Code Insertion

This technique consists of inserting arbitrary instructions into the program without making an impact on the execution of the program. The total number of different programs possible through garbage insertion is limited from above by the total number of free bits available for program space, which is limited only by available memory for program storage and is clearly enormous [11].

Similarly to instructions substitution, described in 2.1.2, the inserted code can be optimized out, therefore it should be applied after optimizing the program and it might get easily removed by a reverse engineer who is analyzing the program.

This technique could be used to bypass simple automated malware analysis engines, as it breaks the signature of the program. The garbage instructions get executed, which adds complexity while performing

7. https://github.com/xoreaxeaxeax/movfuscator
8. https://github.com/jbangert/trapcc

dynamic analysis of the program, but on the other hand, it might impact the performance.

A simple example of this technique is inserting `NOP` instruction into the assembly code. It does not have an impact on the program execution, but it's still reachable by the control flow of the program. A more advanced way to apply this method, described in [11], is to insert spurious calls to the operating system, which could lead the attacker to analyze a large amount of garbage code and increase the time needed to perform the analysis.

Yadegari et al. [12] proposed a generic automated approach for deobfuscation of executable code based on taint analysis, which tracks the flow of values from the program's inputs to its outputs. This method can identify instructions that do not affect the execution of the program and remove the garbage instructions from the code.

2.1.4 Dead code insertion

Dead code insertion is a technique similar to garbage code insertion, described in 2.1.3. The main difference is that the dead code adds a branch to the control flow of the program, but this branch is never taken during the execution of the program.

This method was first introduced by Collberg et al. [3]. The paper suggests, that there is a strong correlation between the perceived complexity of a piece of code and the number of predicates it contains. This technique could be further enhanced by using opaque predicates (2.1.1) – for example, adding a condition with an opaque predicate, which creates a branching point between a valid and a dead branch. The predicate would always evaluate to `True`, making it impossible for the control flow of the program to reach the redundant branch.

Another way to further confuse the reverse engineer, described in [3], is to add dummy blocks of code to the redundant branches. For example, one can clone a sequence of instructions from a valid block of code, introduce a bug into it and place it into the redundant (dead) branch.

This transformation can be removed by utilizing optimization features including in modern compilers, as well as performing the automated deobfuscation approach proposed by Yadegari. et al. [12]. An attack proposed by Salem and Bansescu [13], which is based on machine learning and pattern recognition algorithms to identify obfuscating transformations in the program, might also prove useful to a reverse engineer trying to remove this type of obfuscation.

2.1.5 Mixed Boolean-Arithmetic

A mixed Boolean-arithmetic expression (MBA) is composed of integer arithmetic operations on n-bit words $(+, -, \times, \div)$ and bitwise operations $(\wedge, \vee, \oplus, \neg)$. Zhou et al. [7] present a method to generate an unlimited supply of MBA transforms based on MBA expressions, MBA identities, and invertible functions, which can be used to obscure secret constants, intermediate values, and algorithms, while preserving the original functionality. It is also a useful technique for creating opaque predicates (2.1.2).

The resulting MBA expressions are dependent on program input, so they can not be simplified by a compiler optimization.

There is a practical example in [7] of encoding a constant value $K = 0x87654321$ as a multiterm polynomial MBA function $f(x_1, ..., x_m) = K$ for arbitrary $x_1, ..., x_m$.

The example uses a polynomial

$$f(x) = 727318528x^2 + 3506639707x + 6132886 \pmod{2^{32}}$$

with K as an input, which results in the value 1704256593. Since f is an invertible function, f^{-1} returns the original value of K. To make f^{-1} dependent on the input of the program, the following linear MBA identity is combined with f^1:

$$2y = -2(x \vee (-y - 1)) - ((-2x - 1) \vee (-2y - 1)) - 3$$

Note that the result of the right-hand side does not depend on the value of x, which can be an arbitrary integer from the program.

To further obfuscate the MBA expression, another identity is applied:

$$x + y = (x \oplus y) - ((-2x - 1) \vee (-2y - 1)) - 1$$

Combining these identities, function f and its inverse, results in a very complex expression that is able to produce the original value of K during runtime, but is hard to analyze without debugging the program. The resulting encoding of K can be found in the appendix of [7].

Guinet et al. presented a tool called `arybo` [14], which analyzes the operations performed by MBA expressions at the bit-level. The tool is able to significantly simplify MBA expressions, thus presenting a possibility to circumvent this type of obfuscation.

2.1.6 Control Flow Flattening

This transformation was first described by Wang et al. in [15]. The goal of this technique is to obscure targets of the branches between the basic blocks and thus to make the analysis of a program more difficult.

First, the basic blocks of the function are put on the same nesting level, preceded by a new block, usually referred to as the *dispatcher*. The dispatcher contains code that works as a `switch` statement, used to determine which basic block is going to be executed next. In addition to the dispatcher, a routing variable also needs to be created. Each time one of the original basic blocks terminates its execution, the routing variable is updated and the flow of control is transferred back to the dispatcher, which forwards the control flow to the next basic block, in accordance with the value stored in the routing variable.

The main issue with control flow flattening is finding a way to make the information about the dispatcher, the routing variable, and its updating, difficult to analyze. In its naive implementation, where the routing values are hardcoded during the obfuscation (as in Figure 2.1), it's easy to analyze the code of the basic blocks and reconstruct the original control flow graph.

In [15], Wang et. al suggest the use of global pointers, as in some cases, analysis of pointers can be proven to be NP-hard [16]. In contrast, the authors of [17] propose using one-way functions, which are always hard to analyze.

Another issue with this obfuscation method is the computational overhead it introduces due to additional operations performed by the dispatcher. Johansson et. al [18] proposed a novel method using

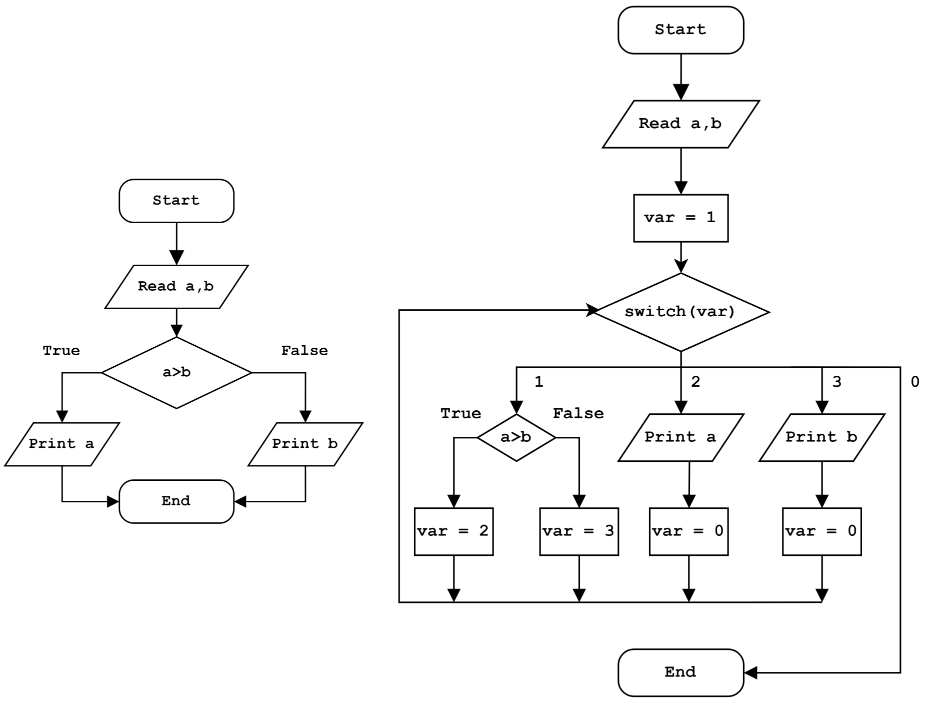

Figure 2.1: Flowchart of a simple program returning a maximum of two numbers, before and after flattening. Notice that the comparison and print statements are on the same level of nesting after being flattened.

lightweight dispatchers, which present a similar level of complexity for the reverse engineer as analyzing a flattened program augmented with cryptographic hash functions, while reducing the overhead by one or more orders of magnitude.

2.2 Evaluating Obfuscating Transformations

Determining the usefulness of an obfuscating transformation is a complex task. When designing and evaluating an obfuscating transformation, one needs to consider multiple criteria, such as how hard would it be for an adversary (e.g., a reverse engineer) to understand the functionality of an obfuscated program P', how hard would it be to construct a deobfuscator, or how much resources would a deobfuscator need to reconstruct the original program P, given P' as an input. Unless the deobfuscation process is fully automated, these criteria will never be fully objective, since they will always, at least partly, depend on the cognitive abilities of the attacker.

Collberg et al. proposed some metrics [3], which can be used to quantify and approximate the quality of obfuscation methods. They have defined the following three criteria:

- *Potency* consists of various metrics which were originally designed to be used to measure software complexity in the field of software engineering, for example, the number of operators and operands in P, number of predicates (cyclomatic complexity) in a function, or a nesting level of conditional statements in a function.
- *Resilience* is measured on a four-point scale, ranging from *trivial* to *one-way*. The value of resilience depends on two parameters – *programmer effort* – how much time would a programmer need to construct a deobfuscator to reduce the *potency* of a transformation, and *deobfuscator effort* – time and space complexity of a deobfuscator which can reduce the *potency* of a transformation. *Programmer effort* is based on the scope of the transformation, from local to inter-process, while *deobfuscator effort* could be either polynomial or exponential.
- *Cost* of a transformation measures how much execution time and space overhead would a transformation introduce to the obfuscated program. This value is also measured on a four-point scale, ranging from *free* (transformation adds a constant overhead) to *dear* (P' requires exponentially more resources than P).

Mohsen and Pinto [19] proposed using Kolmogorov complexity [20] to measure the quality of obfuscation. Kolmogorov complexity can be described as the shortest length of a program, which can produce a given object (e.g., a binary string, or an obfuscated program). Due to the undecidability of the halting problem, exact Kolmogorov complexity can not be computed. However, it can be estimated using compression algorithms. More specifically, Kolmogorov complexity is the lower bound of a compression algorithm.

The idea to use Kolmogorov complexity to quantify the quality of obfuscation is based on an assumption that obfuscation produces irregularities in the obfuscated code (e.g., by inserting opaque predicates, or cloning and diversifying basic blocks), thus making it less comprehensible for the adversary. A program that contains more regular patterns can be compressed with a higher rate, and thus has a lower Kolmogorov complexity. In contrast, an obfuscated program would have higher Kolmogorov complexity, which implies that it is a useful metric for evaluating obfuscation, as shown in [20].

3 LLVM

The LLVM Project started as a research project at the University of Illinois focused on creating a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and idle time between runs [4].

The name LLVM was originally an abbreviation for *Low Level Virtual Machine*. Since the project has little in common with what is currently perceived as a virtual machine, the meaning of the abbreviation has been later removed to avoid confusion, so now LLVM is the full name of the project[1].

The project consists of various sub-projects, such as LLVM Core libraries providing source and target-independent optimizer and code generator, implementation of the C++ standard library with full support for C++11 and C++14, Clang – a compiler for C, C++, and Objective-C, and the LLDB debugger.

3.1 LLVM Compiler Architecture

The LLVM compiler pipeline consists of three main phases. First, the frontend parses the input code, which includes validation of syntax and semantics and diagnosing errors. If the code is valid, the frontend creates a representation of the code in the LLVM Intermediate Representation (IR) format, which is discussed in further detail in the following section.

Various analysis and transformation passes can be run on the IR. Analysis passes are used to gather some higher-order information about the IR without mutating it, e.g., generate the Control Flow Graph. Transformation passes mutate the IR in some way, for example deleting dead code. Transformation passes can use the results generated by the analysis passes.

The part of a compiler that is responsible for these tasks is typically called the *optimizer*. The main goal of the optimizer is to gather infor-

1. `https://llvm.org/`

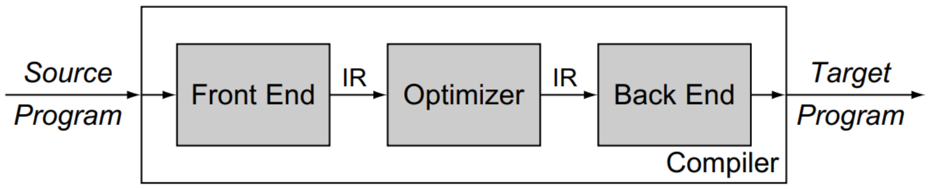

Figure 3.1: A high-level illustration of a three-phase compiler [22]

mation about the runtime behavior of the program and apply it to improve the final code generated by the compiler. The most common goal of the optimizer is to make the program run faster. However, depending on the application, there can be other goals of optimization which would have higher priority. In the case of embedded systems, it could be reducing the size of the compiled code or analyzing the energy consumption of the program in order to make it more energy-efficient[21].

Lastly, the backend takes the IR as an input, executes passes like instruction selection and register allocation, and produces native machine code for the target architecture. This compiler pipeline is usually referred to as a *three-phase compiler*.

The design of the LLVM compiler provides much flexibility, given that the IR is source and target-independent, therefore the optimization passes can be applied while compiling source code of any programming language which has an LLVM frontend. Some of the supported languages are C and C++, Go, Haskell, Rust, and Swift.

LLVM backends, which generate the machine code from the intermediate representation, also support various target architectures, such as x86, ARM, and PowerPC.

For the purpose of obfuscation, the focus of this thesis is the middle section of the pipeline. Even though the optimizer works only with the intermediate representation form, which is passed to the backend afterward, transformations applied in this phase have an impact on the structure of the final executable file.

3.2 LLVM Intermediate Representation

The LLVM IR is a low-level language similar to assembly. However, it provides various abstractions which remove target-specific instructions and features. It uses a set of infinite temporary registers and the *Single Static Assignment* (SSA) form, which means that it needs to fulfill two constraints:

1. Each definition has a distinct name,
2. Each use refers to a single definition.

The single-assignment property of the namespace allows the compiler to sidestep many issues related to the lifetimes of the values, for example, because names are never redefined or killed, the value of a name is available along any path that proceeds from that operation. These two properties simplify and improve many optimization techniques[22].

The Intermediate Representation is organized into modules – each module corresponding to a single source file (also referred to as translation unit). During the compilation process, separate modules are linked with the LLVM Linker, also called LLD, to produce the executable file. Modules generally contain functions and global variables, both of which are called *global values*. Global values are represented by a pointer to a memory location and their identifiers begin with the '@' character. In addition to global value identifiers, there are also local identifiers, prefixed with the '%' character. They are typically used to register names and types.

Listing 1 shows a simple program in LLVM IR, which can be used to further introduce the LLVM IR syntax. On the first line, there is a definition of a global variable, a constant string, named `.str`. It consists of 12 characters, size of each of them is 8 bits. The `private` keyword states that this value is only directly accessible by objects in the current module and `unnamed_addr` indicates that the address is not significant, only the content. Constants marked like this can be merged with other constants if they have the same initializer.

The module defines two functions – `foo` and `main` – starting on lines 3 and 20 respectively. Every function definition starts with the `define` keyword and specifies the function's return type (a 32-bit

```
@.str = private unnamed_addr constant [12 x i8] c"Hello world\00", align 1

define i32 @foo() {
entry:
  %add = add i32 7, 42
  %cmp = icmp sgt i32 %add, 56
  br i1 %cmp, label %if.then, label %if.end

if.then:
  br label %return

if.end:
  br label %return

return:
  %retval.0 = phi i32 [7, %if.then], [%add, %if.end]
  ret i32 %retval.0
}

define i32 @main() {
entry:
  %call = call i32 @puts(i8* getelementptr inbounds([8 x i8], [8 x i8]* @.str, i64 0, i64 0))
  %call1 = call i32 @foo()
  ret i32 0
}

declare dso_local i32 @puts(i8*)
```

Listing 1: Example of a short LLVM IR module. Some parts, such as target architecture information, metadata, and comments, have been omitted for better readability. Also, the compiler optimizations have been disabled.

integer in this case). There is also an external declaration of the `puts` function, which is called in the `main` function to print the "Hello world" string. The main function then performs a call to the `foo` function and terminates by returning zero. The `foo` function consists of multiple basic blocks. Every basic block is prefixed with its name and a colon, and ends with a terminator instruction, such as `br` (branch) and `ret` (return). The function simply adds two integers (7 and 42), compares the result with another integer, and transfers the control flow to the next basic block, based on the result of the comparison.

The SSA form, described at the beginning of this section, introduces a special type of instructions, called `phi` instructions. They appear at the point where different paths of control flow merge, as we can see at line 16 in Listing 1. The `phi` instruction assigns a value to the virtual register named `%retval.0`, depending on the block from which the control entered the current block. If the control entered from the `%if.then` block, value 7 is assigned. Otherwise, the control entered from the `%if.end` block and the value stored in the virtual register `%add` is chosen.

The example code in Listing 1 has been generated and tweaked to present a simple and readable form of the LLVM IR. First, `clang` was used to produce a human-readable form of the IR:

```
$ clang -S -emit-llvm -c example.c -O0 -Xclang
-disable-O0-optnone -fno-discard-value-names
-o example.ll
```

The `-O0` flag tells `clang` to not perform any optimizations on the file. However, to be able to additionaly apply passes using `opt`, we need to include the `-Xclang -disable-O0-optnone` flags. Thanks to the `-fno-discard-value-names` flag, the IR contains meaningful names of the registers and basic blocks, instead of numbers only (e.g., `%1, %2`).

Then the `opt` tool is used to apply the `mem2reg` pass on the IR to transform it into a more simple form by promoting memory references into register references[2]:

```
$ opt -mem2reg -S example.ll -o example.ll
```

2. `https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register`

4 Existing tools

4.1 Obfuscator-LLVM

Obfuscator-LLVM is an open-source project initiated in 2010. It aims to provide increased software security through code obfuscation and tamper-proofing[1]. The project is a fork of the LLVM compilation suite, therefore it works with the LLVM IR, utilizing LLVM's possibility of writing custom transformation passes. It supports all programming languages and target platforms that are currently supported by LLVM.

The open-source version of the project implements three obfuscating transformations:

1. *Instructions Substitution* is the most simple obfuscating transformation included in the project. It replaces simple binary operations with more complex ones, as described in Section 2.1.2. Obfuscator-LLVM supports the substitution of integer additions and subtractions and the Boolean operators AND (&), OR (|) , and XOR (^). For example, the expression $a = b \wedge c$ is substituted as $a = (b \oplus \neg c) \wedge b$. Some of the operations have multiple substitution candidates, which are chosen randomly to increase code diversity. A full list of the implemented substitutions can be found in [23]. The substitutions are rather simple and according to the authors, this transformation can easily be circumvented by re-optimizing the generated code.

2. *Bogus Control Flow* modifies the control flow graph by inserting a new basic block before an existing one. The new basic block ends with an opaque predicate (2.1.1), which always evaluates to `True`, making a conditional jump to the original basic block. The original basic block is also cloned, randomly filled up with various junk instructions, and inserted to the `False` branch leading from the new basic block. This cloned block is never reached, because of the invariant opaque predicate. The weakness in the implementation of this transformation is that it uses just a single

1. Preventing a user from modifying the software against the manufacturer's wishes.

opaque predicate:

$$(y < 10 \,||\, x \cdot (x-1) \mod 2 = 0)$$

The two global variables, x and y, which are declared to construct this predicate, can also give a hint on where the opaque predicates are, which would make it easier for a reverse engineer to overcome this transformation.

3. *Control Flow Flattening* is implemented in a naive way, as described in Section 2.1.6 – by hard-coding the routing values during the obfuscation process. This implementation provides some resilience against automated analysis tools, e.g. signature-based malware scanners, but it does not propose a significant challenge for a reverse engineer or automated tools performing a more complex static analysis.

Apart from the aforementioned obfuscating transformations, this tool also implements a transformation pass for basic block splitting. This pass introduces additional complexity to the transformations manipulating the control flow.

The authors also added a feature to tag specific functions, which are supposed to be obfuscated, in the source code of the program. This way, the developers can reduce the negative performance impacts on the program they are obfuscating, by omitting non-crucial functions from the obfuscation process.

The commercial version of Obfuscator-LLVM used to implement additional, more advanced capabilities, such as code tamper-proofing and procedures merging, mentioned in [23]. However, according to the wiki on the project GitHub repository[2], the commercial version is no longer available since the end of 2016.

The GitHub repository, containing the open-source version, is not being maintained since then as well. The authors of the project apparently founded a startup named Strong.Codes, which has been later acquired by Snap Inc. [24].

2. `https://github.com/obfuscator-llvm/obfuscator/wiki`

4.2 Tigress

Tigress is a source-to-source obfuscator for C language, developed by Collberg[3] as a research project at the University of Arizona. Compared to Obfuscator-LLVM, while supporting only C language, Tigress seems to be a much more powerful obfuscation tool with a wide variety of implemented transformations, such as:

1. *Jitting* (just-in-time compilation) turns the function to be obfuscated into a new one, which generates the machine code of the original function dynamically when it is executed. Another similar transformation supported by Tigress is *JitDynamic*, which continuously modifies and updates the generated code at runtime. This version works at the granularity of basic blocks, decoding them when they need to be executed and subsequently re-encoding them.

2. *Virtualization* turns a function into an interpreter with a custom set of virtual instructions for each function[4]. From the randomly generated instruction set, Tigress generates a bytecode program (array) that is specialized for the obfuscated function. Tigress then selects one of eight dispatch functions (e.g., a switch statement, nested conditional statements, jump statements etc.) and produces an output program. This transformation can be further enhanced by using opaque predicates, inserting bogus code, encoding constant values, and dynamically encoding the array containing the program bytecode, using the same mechanism as *JitDynamic*.

3. *Flattening* implements a slightly more advanced approach to flatten the control flow graph than the transformation implemented in Obfuscator-LLVM. Four kinds of block dispatchers are available – basic implementation using `switch` statement, direct `goto` statements and indirect ones using a jump table, and *call dispatch*, which creates a new function out of every basic block and uses a table of function pointers to jump to them. Obfuscation of the routing variable, which points to the next basic block to

3. The co-author of [3], [5] and others
4. `https://tigress.wtf/virtualize.html`

be executed, can be also enhanced by using opaque predicates (2.1.1).

Furthermore, Tigress also provides transformations like splitting and merging functions, encoding literals, data, arithmetic operations, external calls, and branch targets. To harden the program against static analysis tools based on alias analysis, it replaces direct function calls with indirect ones, storing function addresses in a global array.

Tigress also contains a transformation to prevent taint analysis. To hide a variable from tainting, it inserts a block of code that copies its value in a loop, bit by bit, to a new one. This way, the original tainted variable may be discarded. Currently, this transformation can be applied only on a small set of variables, such as `argc` and `argv` in `main`.

4.3 UPX (Ultimate Packer for eXecutables)

UPX is an open-source software used for packing binary executable files. It is not primarily developed to be used for obfuscation. It creates an executable that is composed of three parts:

1. Compressed executable,
2. Empty section, in which the executable is unpacked during runtime. It has the same size as the original unpacked executable.
3. Stub, which is an entry point of the packed executable. It allocates memory for unpacking the executable during runtime, extracts the original file, resolves imports, restores file permissions, and jumps to the entry point of the original file.

Thanks to its portability and support of a wide variety of executable formats, UPX is popular among malware authors. Since the compression dramatically changes the file signature, packing a malicious executable can be used to fool many automated tools for malware analysis.

However, because of the popularity of UPX in the malware community, executables packed with this software might get automatically flagged as suspicious by antivirus scanners (e.g., by looking for specific strings in section headers – *UPX0, UPX1*). The original program can be also easily unpacked with `upx -d`. One of the simple tricks used by malware authors is to modify the packed file in a way that breaks the unpacking method. For example, by modifying the section header names, the automatic unpacking method will fail, but the stub will still be able to unpack and execute the original binary [25].

Since UPX is open-source, malware authors can always modify the original source code to create a custom version of UPX. It is still possible to obtain the original executable packed by a custom UPX, since it needs to be unpacked at some point during execution, therefore the problem is reduced to finding the unpacking stub and extracting the original file from memory, when it gets unpacked.

To obfuscate malware, UPX is typically used as one of the multiple layers of obfuscation, together with various techniques to prevent debugging and tampering the code.

5 Design

The obfuscating transformations designed for this thesis are built on the foundations from Obfuscator-LLVM, while improving its currently available transformations and writing some new ones from scratch. Thanks to the modularity of the LLVM pass framework, each of the transformation passes can be applied independently, or combined in a sequence.

Selected transformations are have been designed with regard to their resilience against reverse engineering and impact on performance.

5.1 Instructions substitution

While applying basic rewrite rules for binary operations, such as the ones implemented in Obfuscator-LLVM, one can easily deobfuscate the expressions with an optimizing pass included in LLVM's `clang`, for example, the `InstCombine`[1] pass, which is supposed to "Combine instructions to form fewer, simple instructions." However, when the rewrite rules include MBA (2.1.5) expressions applied multiple times, the optimization fails to reconstruct the original expressions, as the experimental results in [26] show.

Based on the method for generating MBA identities proposed by Zhou et al. [7], Eyrolles has generated a list of all rewrite rules composed of three boolean expressions [26]. We have decided to implement a selection of those rules in the obfuscator. For each binary operation, the rewrite rule is picked randomly out of three possibilities. to increase code complexity and diversity. Below are the rewrite rules implemented in the pass.

1. `http://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions`

5.1.1 Rewrite rules

Addition:

$$\begin{aligned} x+y &\to 2(x \vee y) - (x \oplus y) \\ x+y &\to (x \oplus \neg y) + 2(x \vee y) + 1 \\ x+y &\to (x \oplus y) + 2y - 2(\neg x \wedge y) \end{aligned}$$

Subtraction:

$$\begin{aligned} x-y &\to 2(x \wedge \neg y) - (x \oplus y) \\ x-y &\to \neg(\neg x + y) \wedge \neg(\neg x + y) \\ x-y &\to x \wedge \neg y - \neg x \wedge y \end{aligned}$$

XOR:

$$\begin{aligned} x \oplus y &\to (x \vee y) - (x \wedge y) \\ x \oplus y &\to (x \vee y) - y + (\neg x \wedge y) \\ x \oplus y &\to (x \vee y) - (\neg x \vee y) + (\neg x) \end{aligned}$$

AND:

$$\begin{aligned} x \wedge y &\to (\neg x \vee y) - \neg x \\ x \wedge y &\to (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\ x \wedge y &\to -(x \oplus y) + y + (x \wedge \neg y) \end{aligned}$$

OR:

$$\begin{aligned} x \vee y &\to (x \wedge \neg y) + y \\ x \vee y &\to (x \oplus y) + y - (\neg x \wedge y) \\ x \vee y &\to (x \oplus y) + (\neg x \vee y) - (\neg x) \end{aligned}$$

5.2 Opaque constants

To prevent the reverse engineer from extracting constant numerical values from the program, we have decided to implement a method proposed by Zhou and Main [7] to obscure the data by combining MBA identities (2.1.5) with invertible polynomials.

Let:

- f be an invertible polynomial over $\mathbb{Z}/(2^n\mathbb{Z})$
- $g = f^{-1}$
- E an MBA expression non-trivially equal to zero, for example $E = x + y - (x \vee y) - (\neg x \vee y) + (\neg x)$
- C a constant to be obfuscated.

To obfuscate C, we can encode it as $C = g(E + f(C))$. The following example will show how the process works in practice. Suppose we are performing calculations in $\mathbb{Z}/(2^n)$ with 32-bit words[2], f is a linear function with coefficient a being an odd integer, so that it is invertible mod 2^{32} and b is an arbitrary integer, while all integer values are 32 bits large:

$$\begin{aligned}
C &= 123456 \\
a &= 1337 \\
b &= 42 \\
x &= 1307300694 \\
y &= 2583472541 \\
f(x) &= ax + b = 1337x + 42 \\
g(x) &= a^{-1}x + (-a^{-1}b) \\
&= 1337^{-1}x + (-1337^{-1} \cdot 42) \\
&= 1185372425x + 1753965702 \\
E &= x + y - (x \vee y) - (\neg x \vee y) + (\neg x)
\end{aligned}$$

Then we can encode C as:

2. All calculations are $\mod 2^{32}$.

$$\begin{aligned}
f(C) &= 1337 \cdot 123456 + 42 = 165060672 + 42 \\
E &= 1307300694 + 2583472541 - 3724540895 \\
&\quad - 3153898941 + 2987666601 = 0 \\
C &= g(x + y - (x \vee y) - (\neg x \vee y) + (\neg x) + f(C)) \\
&= g(0 + 165060672 + 42) \\
&= 1185372425 \cdot 165060714 + 2541001594 \\
&= 123456 \mod 2^{32}
\end{aligned}$$

The intermediate results of the functions f, g, and the MBA expression E are calculated during runtime, which makes it much harder to obtain the value of C during static analysis. The values of a, b, x, y can be either randomly generated during compilation (obfuscation) time, or by injecting a call to a random number generator that executes during runtime. The third option is to use suitable program inputs or function arguments, which would hinder deobfuscation techniques based on taint analysis[3].

For the purposes of testing and demonstration of this obfuscation method, we have chosen to generate the values of a, b, x, y – by using a random number generator during the compilation time.

5.3 String obfuscation

Strings, i.e. sequences of characters, are present in most of the existing programs. They are used to improve accessibility for the user interacting with the program, or to produce meaningful logging and error messages for the developers. For the reverse engineer, strings can present a valuable source of information about the program. They can be easily viewed in common decompiler and disassembler software, as well as displayed in `bash` by using `strings`[4] command.

LLVM provides a convenient way to obfuscate all strings used in the program. During the obfuscation, the strings can be encoded

3. Tracking flow of values in the program and identifying values and variables that influence program's outputs, as well as the control flow of the program.
4. `https://linux.die.net/man/1/strings`

using an arbitrary function that transforms them. Then, a decoding function is injected into the LLVM IR module. During runtime, the decoding function is used to obtain the original strings. This approach provides resilience against static analysis, while also introducing more complexity when performing dynamic analysis.

The encoding and decoding functions can be different for each obfuscated program, which gives us a choice to either reduce the computational overhead by using an efficient and simple encoding and decoding function, or to strengthen the obfuscation by using more complex functions. When a keyed function is used, the key needs to be included in the obfuscated program. A possible way to harden this transformation is to use other obfuscation methods, such as the one mentioned in the previous section (5.2), to hide the key.

5.3.1 Encoding and decoding functions

For the purposes of hiding the strings, we have designed two simple functions, which can be used either individually, or combined, if we apply the pass multiple times. The first one shifts every character in the string by 13, e.g., it changes every character *a*, represented in decimal as 97, to character *n*, represented in decimal as 110.

The second, more complex function, uses a key, which is represented as a string. It iterates over the characters in the obfuscated string and performs a bitwise XOR, using a character on the corresponding position in the key string as a second operand. If the obfuscated string exceeds the length of the key, the function loops back to the first character of the key string. To add more confusion, this function obfuscates the string terminator (represented by decimal 0) as well.

5.4 Bogus Control Flow

Bogus Control Flow pass is one of the passes included in *Obfuscator-LLVM* (4.1). The weakness of the original implementation is that it uses just a simple opaque predicate with two global variables, both initiated to zero. Any experienced reverse engineer could probably detect this type of opaque predicate by just simply looking at the disassembled program, while automated tools based on symbolic execution would definitely identify the predicate and simplify it.

We have decided to improve this pass by replacing the weak opaque predicate with two types of stronger ones.

5.4.1 Arithmetic opaque predicates

Predicates of this type are composed of mathematical formulas which are always `True` (invariant). They include basic arithmetic operations $(+, -, \times, \div)$ and remainder operations. MBA (2.1.5) is not used here, because running the Instructions Substitution pass would turn the simple operations into MBA anyways, therefore implementing them here as well would result in redundancy and decreased modularity. Following formulas, taken from [27], are included in the pass:

$$\begin{aligned}
&x \neq 7y^2 - 1 \\
&0 \neq (x^2 + 1) \mod 7 \\
&0 \neq (x^2 + x + 7) \mod 81 \\
&0 \neq (4x^2 + 4) \mod 19
\end{aligned}$$

To further harden these predicates, we use input arguments of the obfuscated function for the variables x and y. Since the formulas are always true for an arbitrary integer, the predicate will always lead to a correct block. Using the function arguments in the opaque predicate may prevent various deobfuscation tools and techniques based on taint analysis (e. g. [12]) from detecting and removing the predicate.

When the function does not have suitable arguments, one or two global variables (depending on the number of available integer arguments) are created and initialized to a random value instead.

5.4.2 Symbolic memory opaque predicates

This type of opaque predicate is based on the ideas first proposed by Collberg et al. [5]. The exact method to construct is described in [9]. This is the technique which we have decided to use[5].

5. A deprecated, proof-of-concept implementation of this method can be found at `https://github.com/hxuhack/symobfuscator`

The predicate is constructed by creating two arrays and using an input value of the obfuscated function. The remainder r after the division of the input value by the size of the first array is used as an index to get the rth element from the first array. This element then serves as an index to the second array, which yields an element from the second array that can be used to construct the opaque predicate.

Since the value of the resulting element from the second array depends on the input symbol, detecting and removing this type of opaque predicate presents a hard problem (first described by King [28]) for deobfuscation tools based on symbolic execution.

6 Implementation

One way of integrating new passes using the LLVM framework is to include them in the LLVM source tree and build LLVM from the source to use them during the compilation and obfuscation. The other way is to build an out-of-tree pass, which means to compile the source code of the pass separately, in an arbitrary destination, and load it with the LLVM `opt` tool to apply it on an LLVM IR module. We chose the out-of-tree method, as it is more convenient and straightforward to use.

To make the transformations more diverse, the passes use a random number generator based on the 64-bit Mersenne Twister by Matsumoto and Nishimura (`std::mt19937`), together with the `std::random_device` interface to generate non-deterministic random numbers.

Instructions on how to build and use the implemented passes are included in Appendix A.

6.1 Instructions substitution

Implementation of this pass is based on a tutorial [29] created by developers from Quarkslab, which shows how to write a simple obfuscating transformation that substitutes an addition operation with an MBA expression. The pass iterates over instructions in a basic block, searching for instructions of `BinaryOperator` type that operate with integers. To build a new instruction, the `IRBuilder` class is used.

For each substituted instruction, the pass generates a random number from a uniform integer distribution in a range from 1 to 3 (number of available substitutions for each operation) using the random number generator described at the beginning of this chapter. After the new instructions are generated and their result is stored in a new virtual register, we call the `ReplaceInstWithValue()` function, which replaces all uses of the original instruction in the basic block, and removes the original instruction.

This pass can be applied to the module multiple times, recursively generating more complex MBA expressions with each iteration. Listings 2 and 3 show how a simple instruction – `%add = add i32 %a, %b`

– which performs addition and stores the result in `%add`, gets obfuscated after two iterations of this pass.

```
%2 = xor i32 %a, %b
%3 = or i32 %a, %b
%4 = mul i32 2, %3
%add = sub i32 %4, %2
```

Listing 2: IR code after using the substitution:
$x + y \rightarrow 2(x \vee y) - (x \oplus y)$.

```
%2 = xor i32 %a, -1
%3 = xor i32 %a, -1
%4 = or i32 %3, %b
%5 = or i32 %a, %b
%6 = sub i32 %5, %4
%7 = add i32 %6, %2
%8 = or i32 %a, %b
%9 = mul i32 2, %8
%10 = sub i32 0, %9
%11 = add i32 %10, %7
%12 = sub i32 0, %11
%13 = sub i32 0, %9
%14 = add i32 %13, %7
%15 = sub i32 0, %14
%add = and i32 %15, %12
```

Listing 3: IR code from Listing 2 after substituting:
$x \oplus y \rightarrow (x \vee y) - (\neg x \vee y) + (\neg x)$
$x - y \rightarrow \neg(\neg x + y) \wedge \neg(\neg x + y)$

6.2 Opaque constants

The boilerplate code for this pass is from the Quarkslab developers' blog [30]. However, the core principles and scope of obfuscation are fundamentally different. As described in Section 5.2, this pass obfuscates unsigned 32-bit integer values.

The main loop iterates over instructions of a basic block, ignoring those which are either a call to a function, switch, or a GEP[1] instruction. When a suitable instruction is found, the pass checks whether some of its operands are constant 32-bit positive integers. Afterward, the program proceeds to build an expression described in Section 5.2. To find the modular inverse of a, `boost::integer::mod_inverse()` function from the Boost[2] library is being used.

Some of the values, most importantly, the multiplication of the obfuscated constant by the coefficient a in function f, are computed during compilation time. Other parts, such as the expression E and composition of functions f and g, are created using the `IRBuilder`, which is instantiated with the first template argument being `NoFolder`. This template argument specifies a class used for constants. The default option – `ConstantFolder` – performs constant folding during compilation time, which could possibly weaken this obfuscating transformation, therefore `NoFolder` is used.

6.3 String obfuscation

Implementation of this pass is partly based on a tutorial [31] for building an LLVM pass for string obfuscation. The pass iterates over global variables in the IR file, since all strings in LLVM IR, are stored as global variables. When a global variable that contains a string is encountered, it is encoded, so it does not appear in its original form in the resulting binary.

Both encoding and decoding functions have to be provided in a separate LLVM IR module, named `codec.bc`. They have to be named

1. Get Element Pointer – used to get the address of a subelement of an aggregate data structure.
2. `https://www.boost.org/`

`encode` and `decode` and their arguments need to be of type `unsigned char *`.

For each encoded string, a call to a decoding function is created and inserted into the IR file. More specifically, a `decode_stub()` function, which contains a call to a decoding function for every string, is injected at the beginning of the `main()` function in the obfuscated program. This approach has some disadvantages. At some point during the execution of the program, all of the strings will appear decoded. However, this transformation is still useful to hinder static analysis, as well as dynamic analysis, especially when the decoding function is obfuscated with other transformations. The second disadvantage is that the obfuscated program needs to have a `main()` function, otherwise the `decode_stub()` function can not be injected. This second issue could possibly be solved by searching for other entry points of a program, apart from `main()`.

The main advantage of our implementation is that the user can define arbitrary encoding and decoding functions (but also needs to ensure they are working correctly[3]). Both functions are first loaded from a separate LLVM module (bitcode file) using the `ParseIRFile()` function. Then, the functions can be stored in an object of `Function` type. While the `decode()` function is simply injected into the IR module, the `encode()` function is directly executed during the obfuscation, by utilizing the `ExecutionEngine` abstract interface.

To harden this obfuscation pass, we tell the compiler to always inline the decode function, by adding the `AlwaysInline` attribute to it in the pass. This modification does not appear explicitly in the bitcode, since the inlining will be performed by the backend and the result is observable only in the compiled program.

6.4 Bogus Control Flow

The code for splitting and cloning the basic blocks is taken from the original Bogus Control Flow pass in Obfuscator-LLVM, with just a slight modification.

3. For example when obfuscating a program written in a language with different terminating characters than `'\0'`.

The original code had a bug in function `AddBogus()`, which is responsible for splitting the basic blocks, creating new branches, and inserting placeholder instructions, which are later replaced by an opaque predicate. The bug is related to the exception handling mechanism in LLVM IR – when a basic block might throw an exception, the terminator instruction at the end of the block is the `invoke` instruction, which can either lead to a regular basic block, or a special basic block that starts with the `landingpad` instruction and handles the thrown exception. The label of this block is marked with `unwind` keyword.

However, when the block with the `landingpad` is obfuscated, a new block, which starts with the opaque predicate, is inserted before the original one. This causes a situation when the `unwind` label leads to a block that does not start with the `landingpad` instruction, generating an invalid IR module. Also, the new block with the `landingpad` instruction does not have an incoming edge marked as `unwind`, coming from an `invoke` instruction. Both of those facts result in a compilation error.

Luckily, the `VerifierPass` was able to detect this erroneous behavior and display a corresponding error message. To fix this issue, we have added a check at the beginning of the `AddBogus()` function, to find out whether a block contains a `landingpad` instruction so that such blocks can be skipped and not obfuscated.

Our main contribution to this pass are the `getSimpleOP()` and `getSymOP()` functions, which construct the opaque predicates that are described in Section 5.4. First, the pass iterates through the obfuscated function arguments and stores those which are 32-bit integers.

If such arguments are found, they are used to create an opaque predicate. In the `getSimpleOP()` function, the values of the obfuscated function arguments are divided by INT8_MAX (127) and the remainders are used as the values for x and y in the formulas. This way, we prevent a situation when a result of some of the operations might overflow the type of the value where it is being stored, while we also reduce the computational overhead of the obfuscated program. One of the four formulas is then picked randomly to construct the predicate, which is inserted into the basic block afterward.

The implementation of the `getSymOP()` function is more complex. First, it allocates two arrays of integers on the stack and initializes their values (a range from 0 to array size minus one). Then, a remainder of a division of the input argument by the size of the first array is stored in a virtual register by creating a `srem` (signed remainder) instruction. This register is then used as the second index in a `getelementptr`[4] instruction, indexing the first array. An element pointed to by the result of this `getelementptr` instruction is then stored into a new virtual register, which is used as an index for a second `getelementptr` instruction, indexing the second array.

Loading an element from the second array like this allows us to construct an invariant opaque predicate by creating an `icmp` (integer comparison) instruction with the two operands, one being the loaded element, the other one is the size of the arrays. Since the values stored in the arrays are not larger than the size of the arrays, this comparison has an invariant result and can be used as an opaque predicate.

Listing 4 shows a snippet of the generated IR code that performs described operations (initialization of the arrays `%arr1` and `%arr2` is omitted). The obfuscated function is `main()` with standard arguments `(int argc, char *argv[])`, therefore the symbolic value used in this case is `%argc`.

```
%allocaInst = alloca i32
store i32 %argc, i32* %allocaInst
%loadInst = load i32, i32* %allocaInst
%load64 = sext i32 %loadInst to i64
%0 = srem i64 %load64, 8
%idx_1 = getelementptr inbounds [8 x i64], [8 x i64]* %arr1, i32 0,
 ↪  i64 %0
%1 = load i64, i64* %idx_1
%idx_2 = getelementptr inbounds [8 x i64], [8 x i64]* %arr2, i32 0,
 ↪  i64 %1
%2 = load i64, i64* %idx_2
%ArrOpq = icmp ne i64 %2, 8
```

Listing 4: IR code for constructing symbolic opaque predicates.

4. `https://llvm.org/docs/GetElementPtr.html`

6.5 Limitations and possible extensions

The passes have been implemented as a proof-of-concept for demonstration and testing purposes. Following ideas can be implemented to improve the project and bring it closer to a form of a usable product.

- To make the obfuscation more user-friendly, a scheduling pass can be implemented, which would run individual passes, together with LLVM optimization passes, in a sequence specified by the user.
- More control over the obfuscation process can be provided to the user by introducing additional command-line options, e.g., to control the extent of each transformation, or to specify what types of opaque predicates shall be used.
- For deterministic results, a random number generator that could be seeded with a value provided by the user can be used instead of the current RNG.
- For real-world projects, it would be useful to allow the user to annotate functions that are supposed to be obfuscated. Obfuscating only a selected subset of functions in a project would probably lead to a smaller overhead from the obfuscation.
- To further improve the security of the binary against reverse engineering, redundant information, such as debugging symbols, can be removed by using the `llvm-strip`[5] tool, or its GNU alternative.

5. `https://llvm.org/docs/CommandGuide/llvm-strip.html`

7 Testing and Evaluation

In this chapter, we are going to evaluate the obfuscation techniques based on the criteria and metrics described in Section 2.2 and test the obfuscation on selected samples of code. The test programs consist of a C++ implementation of SHA-512 hash function[1], a C++ implementation of AES-CBC[2] with 128-bit key, and an implementation of QuickSort[3] algorithm in C.

7.1 Potency

For testing potency, we are going to compute the software complexity metrics of the obfuscated bitcode, and compare it to the non-obfuscated version.

Except for the Bogus Control Flow pass, the transformations do not significantly change the Control Flow Graph of the program, since the passes obfuscating constants and substituting instruction operate only in the scope of individual basic blocks, while the pass obfuscating strings manipulates global variables in the bitcode. It also injects a call to decode the string, but this operation does not change the CFG in a significant way. Therefore, we are going to use instruction count (μ_1) and Kolmogorov complexity (μ_2) as metrics to evaluate the potency of obfuscations, and avoid other metrics which are mostly influenced by the changes of the CFG.

To obtain the values, we use the `obfuscation-metrics` tool[4], which includes an implementation of a simple LLVM analysis pass that parses the IR module and outputs the metrics. To compute Kolmogorov complexity, the tool uses the `zlib::compress()` function, which is included in the LLVM libraries.

Table 7.1 shows the changes in software complexity metrics after applying obfuscation, which allows us to estimate the potency of the transformations. Values in the table show a ratio of metrics of an obfuscated and a non-obfuscated program. Substitution $\times 2$ shows

1. `https://github.com/martynafford/sha-2`
2. `https://github.com/kkAyataka/plusaes`
3. `https://www.programiz.com/dsa/quick-sort`
4. `https://github.com/b-mueller/obfuscation-metrics`

the effects of applying the Instruction Substitution pass twice. String obfuscation 1 and 2 refer to the two different string encoding functions – based on bit rotation and bitwise XOR, respectively. The last four rows show the changes of the metrics after applying multiple passes sequentially. The ordering of the passes is discussed in detail in Section 7.1.2.

7.1.1 The potency of individual passes

Instructions substitution

The Instructions substitution pass has been applied to the test program in two iterations. Comparing the change of the metrics in the first and the second iteration, we can see that applying the transformation multiple times does increase μ_1 more than μ_2. This result implies that while the size of the program grows larger with each application of this pass, some regular patterns probably start to appear in the obfuscated program.

Opaque constants

The results for the pass creating the opaque constants show that it has a relatively high potency. Structure and other specifics of the obfuscated programs might be a factor that contributes to this result. Even though we have used only one pair of functions to build the expressions for this obfuscation (as described in Section 5.2), this pass inserts a substantial amount of new instructions, while also generating random operands for some of them, resulting in a significant increase of both metrics.

String obfuscation

String obfuscation comes out as the least potent transformation, but this result might be caused by the low number of strings in the obfuscated programs. Slightly higher potency in the case of using the string encoding and decoding functions based on bitwise XOR is caused by these functions being more complex, compared to the ones based on the rotation of bits.

Bogus control flow

Bogus Control Flow is the most potent pass, as it not only injects new instructions, but also duplicates the basic blocks and creates randomly initiated global variables. The significant increase of μ_1 is probably caused by the creation of the symbolic memory opaque predicates (Section 5.4.2), as it inserts an `alloca` instruction for each element of the arrays created in this process. On the other hand, metric μ_2 might be mostly influenced by the creation of the randomized global variables, similarly as in the case of the Opaque constants transformation.

7.1.2 Combining the passes

After evaluating each pass individually, we have determined the following sequence in which we apply all of the passes:

String obfuscation[5] $\rightarrow$ *Bogus* $\rightarrow$ *Opaque constants* $\rightarrow$ *Substitution.*

The reasoning behind choosing this order of the passes is following:

- The String obfuscation pass injects new functions, which are not obfuscated, therefore it is reasonable to apply this pass first, so the subsequent passes can obscure these functions.
- Bogus Control Flow pass creates clones of the basic blocks, which would be easily identifiable without subsequent obfuscation. The opaque predicates are hardened and diversified by applying the substitution pass.
- Expressions for hiding the constants also benefit from being applied before the substitution pass, same as the opaque predicates.
- Instructions substitution pass has basically the smallest scope – transforming single instructions. Since it might improve the resilience and potency of all the other passes, it is applied last.

5. We have used the encoding and decoding functions based on bitwise XOR, since they have proved to have a higher potency.

Table 7.1 shows that a combination of the passes results in significantly higher potency. We also present the values of μ_1 and μ_2 when the passes have been applied in a reverse order, where it is clearly visible how the particular ordering of the passes can influence the potency. In this case, the values are significantly lower than the original sequence.

Due to the randomness of the obfuscation (e.g., randomly selecting the instruction substitution, randomly generating values for opaque constants), there is a slight variance in the resulting metrics when comparing obfuscated codes originating from the same program, after applying the same transformation in multiple independent runs. However, the variance was only around 4%, therefore we do not include the individual results of multiple independent applications of the transformations in the table.

There is a noticeable pattern among all of the results in Table 7.1. The number of instructions (μ_1) always increases more than Kolmogorov complexity (μ_2). This implies that the transformations have a larger influence on the size of the program, but they do not add irregularities to the program at the same rate. We suppose that this ratio can be made more even, if we add more diversity to the transformations, for example by implementing more substitution candidates for the instructions, creating arrays of various sizes for the symbolic memory opaque predicates, or by inserting random sequences of junk instructions into the cloned blocks in the Bogus Control Flow pass.

Table 7.1: Changes in software complexity metrics after applying obfuscation. Values show a ratio of metrics of an obfuscated and a non-obfuscated program.

Program	SHA-512		AES		QuickSort	
Metric	μ_1	μ_2	μ_1	μ_2	μ_1	μ_2
Substitution	1.37	1.07	1.09	1.04	1.23	1.03
Substitution $\times 2$	2.25	1.26	1.34	1.13	1.82	1.11
Opaque constants	2.66	1.86	3.29	2.40	3.76	1.96
String obfuscation 1	1.05	1.04	1.01	1.02	1.19	1.12
String obfuscation 2	1.09	1.05	1.02	1.03	1.35	1.16
Bogus Control Flow	3.52	2.23	4.28	3.49	7.18	3.40
Str 2 + Bogus	3.72	2.35	4.40	3.64	10.64	4.44
Str 2 + Bogus + Const	14.26	8.08	19.16	16.23	32.43	12.27
All	26.98	11.03	36.05	23.12	57.88	16.23
Reverse order	8.10	4.24	9.50	7.05	21.57	7.84

7.2 Resilience

Resilience is a metric that reflects the difficulty of creating an automatic deobfuscator and its time and space requirements for deobfuscating a program. Since implementing a deobfuscator is a time-consuming task and the results would be vague and subjective (depending on the abilities of the reverse engineer creating the deobfuscator), we will only discuss specifics of the transformation in the context of creating a hypothetical automatic deobfuscator.

Instructions substitution

We consider Instructions substitution pass to have high resilience, since there are no fully automated tools that could reliably simplify MBA expressions. This is mostly due to the fact that there is no clearly defined scale to measure what expressions are easy or hard to comprehend by humans. To implement a deobfuscator, a reverse engineer would first need to identify all of the rewriting rules used to obfuscate the binary operations, and then it might be possible to use *pattern matching* to restore the original instructions. Security of MBA expressions against reverse engineering is further discussed in [7], where the authors experimented with analytical math tools (Mathematica and Maple) in order to prove their assumption that MBA expressions can not be automatically simplified.

Opaque constants

Opaque constants are created using the MBA expressions as well, therefore this transformation has a similar degree of resilience as Instructions substitution. However, in our implementation, we have used just a single MBA identity combined with linear functions, so this pattern might be identified easily, especially when used without any subsequent obfuscation. The resilience (as well as potency) of this transformation could be increased by implementing more MBA identities and using a variety of linear and polynomial functions to construct the expressions.

String obfuscation

String obfuscation might be easily identifiable, since the calls for decoding the strings at runtime are injected into the `main()` function. Therefore we consider the resilience of this transformation to be weak. However, the resilience may also vary depending on the `decode` function used. Since the strings do not appear in the obfuscated binary, this transformation is still resilient against simple tools that perform static analysis, or automated tools running in an environment where they do not have permission to execute binaries.

Bogus Control Flow

In Bogus Control Flow pass, we have focused on creating resilient opaque predicates. To construct arithmetic opaque predicates, we use either the function input values or create global variables. In the case of global variables, the opaque predicate might be easily identifiable and removed during static analysis. We have also tested the resilience of the symbolic memory opaque predicates with an `angr` script (Listing 7.2) on a simple *crackme* program, which takes a string argument from the standard input, compares it with a hard-coded string, and outputs the result. Surprisingly, `angr` was able to find the solution within seconds, despite the obfuscation. However, detecting opaque predicates and removing bogus basic blocks might still be a challenge for a deobfuscator based on symbolic execution, as the authors of [9] argue. Generally, we consider this transformation to be relatively weak in resilience.

```python
def solve(elf_binary):
  project = angr.Project(elf_binary)
  arg = claripy.BVS('arg',8*4)
  initial_state = project.factory.entry_state(args=[elf_binary,arg])
  simulation = project.factory.simgr(initial_state)
  simulation.explore(find=is_successful)
  if len(simulation.found) > 0:
    print(simulation.found[0].solver.eval(arg,cast_to=bytes))

def is_successful(state):
        output = state.posix.dumps(sys.stdout.fileno())
        if b'Correct' in output:
                return True
        return False
```

Listing 5: A snippet of the `angr` script used to solve a simple `crackme`. The script has been adapted from [32].

7.3 Cost

Program obfuscation always requires a compromise between the quality, or strength of the transformations, and its impact on the program performance. In this section, we are going to use the test programs to evaluate how much our transformations impact their computation time by using the Bash `time`[6] built-in utility. Particularly, we are going to measure the total number of CPU seconds that the process spent in user mode.

6. `https://man7.org/linux/man-pages/man1/time.1.html`

For SHA-512 and AES test programs, we have generated random input bytes with `openssl rand`. In each iteration, we have stored the input as a file and passed it to the test program. From figures 7.1 and 7.2, it is visible that the computation time increases with the size of the input. The graphs also show a relation between the performance overhead and potency of the obfuscation – applying all transformations in the order defined in Section 7.1.2 resulted in the slowest computation time, while obfuscating strings and substituting instructions with MBA expressions had the smallest performance impact.

While testing the QuickSort program, we have discovered that obfuscating this program with the Bogus Control Flow pass and executing it on an array containing more than 12370 elements results in a stack overflow. After applying all of the transformations, the program could not handle more than 5550 elements in the array. For this reason, we have excluded the Bogus Control Flow pass from Figure 7.3.

We believe that this issue is caused by the creation of symbolic memory opaque predicates, which involves allocating two arrays on the stack for each predicate, combined with the recursive nature of the QuickSort algorithm. A possible fix for this issue would be to utilize the LLVM Pass Framework to detect recursive functions in a module and use this analysis to avoid inserting symbolic memory opaque predicates into such functions.

We have also noticed that without the Bogus Control Flow pass, applying the passes in the order *String obfuscation – Opaque constants – Instructions substitution* resulted in better performance of the obfuscated program, than when this order has been reversed. We have tested the same combinations of passes (omitting the Bogus Control Flow pass) and the results were similar – worse performance in case of the reverse order. However, the metrics μ_1 and μ_2 have been slightly higher ($\sim 15\%$) for the programs obfuscated with the original ordering of the transformations. This leads us to a conclusion that the cost of a transformation does not always grow proportionally to its potency.

Another important finding from measuring the computation time is that the performance impact of all tested transformations is *constant* in regard to the input size, which can be seen in the Figures 7.1, 7.1 and 7.3.

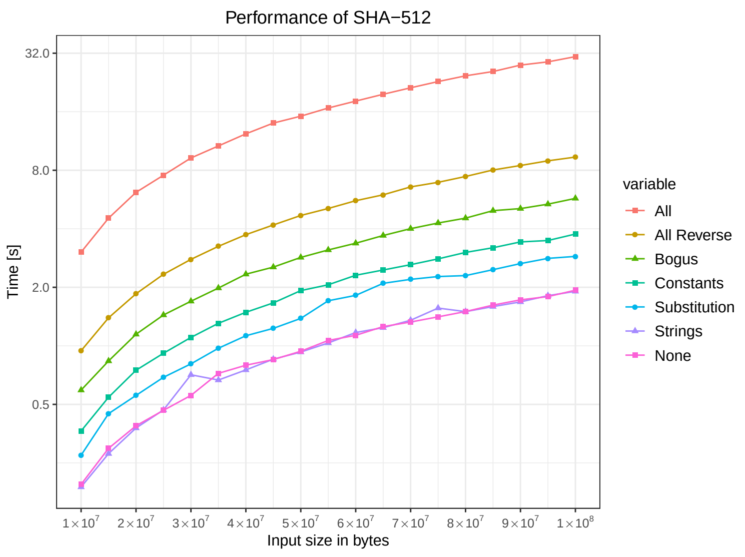

Figure 7.1: Performance of a program computing the SHA-512 hash function. The passes plotted as *All* and *All Reverse* have been applied in the order described in Section 7.1.2.

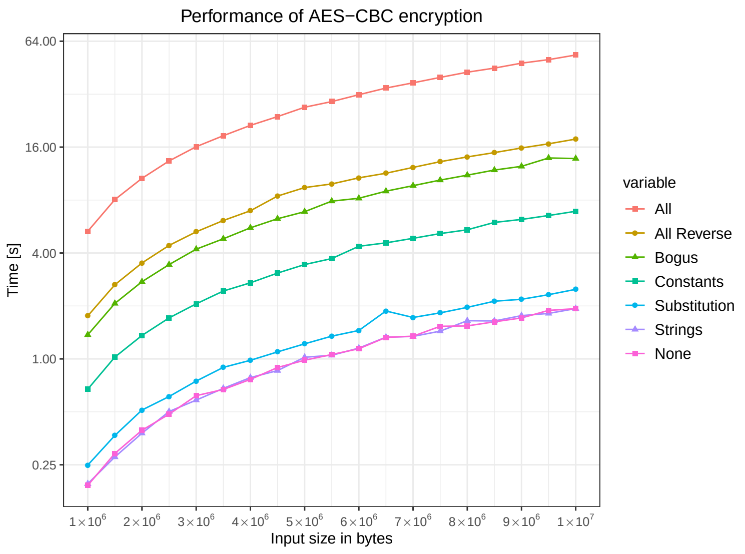

Figure 7.2: Performance of a program performing AES encryption with a 128-bit key in CBC mode. The passes plotted as *All* and *All Reverse* have been applied in the order described in Section 7.1.2.

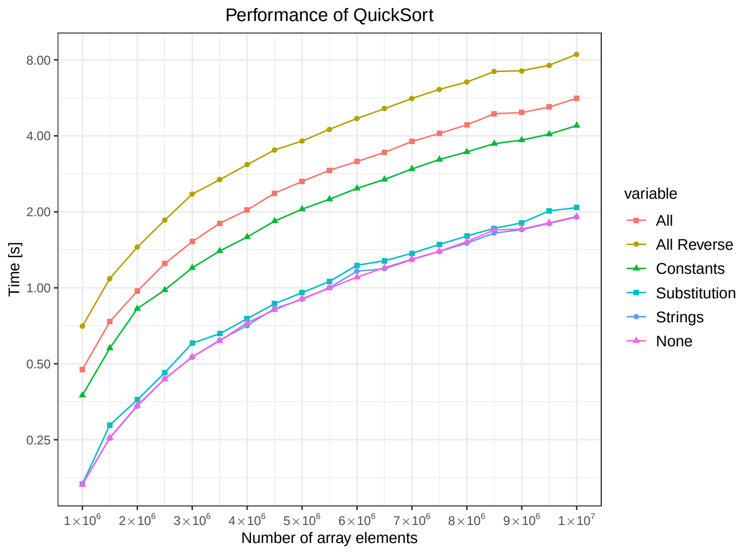

Figure 7.3: Performance of a QuickSort algorithm. The passes plotted as *All* and *All Reverse* have been applied in the order described in Section 7.1.2, except for Bogus Control Flow pass, which has been omitted.

8 Obfuscation on Android OS

Android applications are usually written in Java, which is not supported by the LLVM framework. However, developers have an option to create certain parts of an application using C and C++, for example, to include high-performance libraries in games and other computationally intensive applications. It also allows the developers to reuse various C and C++ libraries or to manage native activities and access physical device components, such as sensors and touch input. The native libraries can be packaged into an Android application package file (APK) using the Gradle build system. The Java code can then call functions in the native library through the Java Native Interface (JNI) framework[1].

Another advantage of using native libraries on Android is the possibility of using an obfuscator based on LLVM to obscure critical components of the application and make it infeasible for potential attackers to reverse engineer them. For example, Snapchat uses this approach to secure a binary that generates the `X-Snapchat-Client-Auth-Token`, which is used in the authentication process between the client and the server in each request [33].

To test whether our passes can be used for Android software, we have created an Android Virtual Device (AVD) with Android 7.1.1 (API version 25), that emulates an ARM Application Binary Interface.

For testing purposes, we have used a modified version of the QuickSort implementation that prints the sorted array. Since Android has a Linux kernel, we can execute compiled binaries without creating a full Android application.

Using the default LLVM tools does not produce executable files compatible with the Android system running on ARM architecture. However, the Android Native Development Kit (NDK) includes prebuilt versions of LLVM tools like Clang, which are needed to produce a program that can be executed in the AVD. We have used `android-ndk-r23b`, which is the latest version of the Android NDK.

To emit the LLVM bitcode, we have used `armv7a-linux-androideabi16-clang`, which can be found after unpacking the NDK zip at `toolchains/llvm/prebuilt/linux-x86_64/bin/`. Using this

1. `https://developer.android.com/ndk/guides`

```
clang -S -emit-llvm qs.c -o qsort_arm.ll
clang -S -emit-llvm qs_main.c -o qs_main_arm.ll
llvm-link qs_main_arm.ll qsort_arm.ll -o qs_arm_linked.ll
opt -load libObfConstPass.so -obfconst -S qs_arm_linked.ll -o
↪  qs_arm_const.ll
clang -fPIE -pie qs_arm_const.ll -o qs_link_const
adb push qs_link_const /data/local/tmp
adb root
adb shell chmod +x /data/local/tmp/qs_link_const
adb shell /data/local/tmp/qs_link_const 10
```

Listing 6: The sequence of shell commands to obfuscate the QuickSort program, compile it for ARM, and execute in the AVD.

version of Clang, we can generate the IR bitcode as described in Section 3.2. Afterward, we can use an obfuscating pass, which is loaded and applied using the `opt` tool from the LLVM framework. Lastly, we create the resulting binary by using the NDK Clang again. In this step, we need to run Clang with `-fPIE` and `-pie` flags, to ensure that the resulting program will be a *Position-Independent Executable* file (PIE).

To test the program in the Android OS environment, we pushed the binary to the AVD using `adb push`. When the executable is pushed to `/data/local/tmp` and its permissions are set correctly, it can be executed and we can see the output in `adb shell`.

If we are compiling multiple source files to create a single executable, we use `llvm-link`, which is also included in the NDK package.

Following the procedure described above, we have tested the obfuscating passes. All obfuscated versions of QuickSort produced a sorted array as an output. That proves that this form of obfuscation is also usable on Android devices. To use the obfuscation with bigger projects, or to produce native libraries and include them in an APK, this process can be automated by integrating it with a suitable build system (e.g. *CMake*).

9 Conclusion

In this work, we have explored obfuscation techniques that can be used to increase the security of compiled code against reverse engineering and other methods of analysis. Currently, a large amount of obfuscating transformations exists. We believe that the transformations described in this thesis are among the most used ones, as they (or their variants) can be found both in obfuscation tools we have analyzed, as well as in multiple research papers that deal with obfuscation or the countermeasures against it.

We conclude that the use of Mixed Boolean-Arithmetic expressions is relevant and has a big potential for designing obfuscating transformations. Regarding manipulation of the control flow of a program, the most important element is the design of resilient opaque predicates. Reverse engineers currently tend to leverage the capabilities of various tools based on symbolic execution and taint analysis, which aid them with understanding an obfuscated program. Therefore, the main focus of the research in the area of obfuscation is shifting towards countermeasures against such tools.

We have also identified and analyzed three freely available tools which can be used for obfuscation. Obfuscator-LLVM has only a small set of capabilities and we reflect that the transformations offered by this tool are rather weak in potency and resilience. However, it provides a basis for implementing obfuscators based on the LLVM Framework. In contrast, Tigress offers various advanced transformations, but a big disadvantage of this tool is that it is limited only to programs implemented in C, since it performs the transformations on the source code level. UPX has the advantage that it works on the machine code level, but since its main purpose is not obfuscation, its capabilities are rather limited. Despite this fact, it is a commonly used tool for obfuscating malware.

Based on the transformations and tools we have examined, we have designed implemented four obfuscating passes using the LLVM Framework. For two of the passes, we have used the MBA expressions, since we believe that they offer a great deal of potency and resilience, while not affecting the performance of a program in a significant way. They provide a convenient way to make the program more complex by substituting simple instructions and encoding constants, while they also improve the qualities of other transformations.

For the implementation of the Bogus Control Flow pass, we have used the source code from Obfuscator-LLVM to manipulate the basic blocks and branches, but we have replaced the original opaque predicate with two types of more advanced predicates. The implementation of String obfuscation pass provides an easy way to create and inject custom encoding and decoding functions into the program, and thus offers a great potential for various extensions.

We have evaluated the transformations in terms of potency, resilience, and cost. The results show that an increase in the size of the program (in terms of instructions count) does not always increase its perceived complexity[1], therefore measures to increase the code diversity, such as the use of randomness, are crucial. Based on the results, we also conclude that applying the transformations in a different order influences the resulting program in a significant way, i.e. different ordering of the passes results in different degrees of potency and performance impact. Since high potency does not always correlate with high performance overhead, we suggest that the ordering of the passes should always be based either on experimenting, or specific heuristics with respect to the nature of the transformations, to achieve optimal results. The cost of the implemented transformations has been proved to be constant – the size of the input increases the computation time of the obfuscated program proportionally to the increase with the non-obfuscated version.

Lastly, we have successfully tested our implementation on a program compiled for Android OS. However, since the transformations manipulate the program drastically, specific cases when the obfuscated program fails in this environment on ARM architecture might emerge. We would like to emphasize that obfuscation on the bitcode level is a complex process and using any transformations for real-world projects always requires rigorous testing.

We reckon that there exist multiple commercially available obfuscating tools that provide more advanced transformations than the ones described and implemented in this work. However, we believe that this thesis can serve as a good starting point, or a reference, for

1. In other words, how it complicates the task of reverse-engineering the program.

other researchers, developers, and reverse engineers interested in this area. Together with the implementation details, it can aid with developing and experimenting with new obfuscating transformations, and help to spark innovative ideas.

Bibliography

1. *Denuvo Runs Into Problems Again, Games Were Unplayable This Weekend - theGeek.games*. 2021. Available also from: `https://thegeek.games/2021/11/09/denuvo-irdeto-drm/`.
2. BARAK, Boaz; GOLDREICH, Oded; IMPAGLIAZZO, Russell; RUDICH, Steven; SAHAI, Amit; VADHAN, Salil; YANG, Ke. On the (Im)possibility of Obfuscating Programs. *IACR Cryptology ePrint Archive*. 2001, vol. 2001, p. 69. ISBN 978-3-540-42456-7. Available from DOI: `10.1145/2160158.2160159`.
3. COLLBERG, C.; THOMBORSON, C.; LOW, Douglas. A Taxonomy of Obfuscating Transformations. In: 1997.
4. LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, 2004.
5. COLLBERG, Christian; THOMBORSON, Clark; LOW, Douglas. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In: San Diego, California, USA: Association for Computing Machinery, 1998, pp. 184–196. POPL '98. ISBN 0897919793. Available from DOI: `10.1145/268946.268962`.
6. AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
7. ZHOU, Yongxin; MAIN, Alec; GU, Yuan X.; JOHNSON, Harold. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In: Jeju Island, Korea: Springer-Verlag, 2007, pp. 61–75. WISA'07. ISBN 354077534X.
8. RAMALINGAM, G. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 1994, vol. 16, no. 5, pp. 1467–1471. ISSN 0164-0925. Available from DOI: `10.1145/186025.186041`.

9. XU, Hui; ZHOU, Yangfan; KANG, Yu; TU, Fengzhi; LYU, Michael. Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 666–677. Available from DOI: 10.1109/DSN.2018.00073.

10. WARREN, Henry S. *Hacker's Delight*. 2nd. Addison-Wesley Professional, 2012. ISBN 0321842685.

11. COHEN, Frederick B. Operating System Protection through Program Evolution. *Comput. Secur.* 1993, vol. 12, no. 6, pp. 565–584. ISSN 0167-4048. Available from DOI: 10.1016/0167-4048(93)90054-9.

12. YADEGARI, Babak; JOHANNESMEYER, Brian; WHITELY, Ben; DEBRAY, Saumya. A Generic Approach to Automatic Deobfuscation of Executable Code. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 674–691. Available from DOI: 10.1109/SP.2015.47.

13. SALEM, Aleieldin; BANESCU, Sebastian. Metadata Recovery from Obfuscated Programs Using Machine Learning. In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. Los Angeles, California, USA: Association for Computing Machinery, 2016. SSPREW '16. ISBN 9781450348416. Available from DOI: 10.1145/3015135.3015136.

14. GUINET, Adrien; EYROLLES, Ninon; VIDEAU, Marion. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In: *GreHack 2016*. Grenoble, France, 2016. Proceedings of GreHack 2016. Available also from: https://hal.archives-ouvertes.fr/hal-01390528.

15. WANG, Chenxi; DAVIDSON, Jack; HILL, Jonathan; KNIGHT, John. Protection of software-based survivability mechanisms. In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 193–202.

16. LANDI, William. Undecidability of Static Analysis. 1992, vol. 1, no. 4, pp. 323–337. ISSN 1057-4514. Available from DOI: 10.1145/161494.161501.

17. CAPPAERT, Jan; PRENEEL, Bart. A general model for hiding control flow. In: *Proceedings of the tenth annual ACM workshop on Digital rights management*. 2010, pp. 35–42.

18. JOHANSSON, Björn; LANTZ, Patrik; LILJENSTAM, Michael. Lightweight dispatcher constructions for control flow flattening. In: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*. 2017, pp. 1–12.

19. MOHSEN, Rabih; PINTO, Alexandre Miranda. Evaluating obfuscation security: A quantitative approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016, vol. 9482, pp. 174–192. ISBN 9783319303024. ISSN 16113349. Available from DOI: `10.1007/978-3-319-30303-1_11`.

20. KOLMOGOROV, A N. On tables of random numbers. *Theoretical Computer Science*. 1998, vol. 207, pp. 387–395.

21. GRECH, Neville; GEORGIOU, Kyriakos; PALLISTER, James; KERRISON, Steve; MORSE, Jeremy; EDER, Kerstin. Static Analysis of Energy Consumption for LLVM IR Programs. In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. Sankt Goar, Germany: Association for Computing Machinery, 2015, pp. 12–21. SCOPES '15. ISBN 9781450335935. Available from DOI: `10.1145/2764967.2764974`.

22. COOPER, Keith D.; TORCZON, Linda. *Engineering a compiler /*. 2nd ed. Amsterdam ; Elsevier/Morgan Kaufmann, 2012.

23. JUNOD, Pascal; RINALDINI, Julien; WEHRLI, Johan; MICHIELIN, Julie. Obfuscator-LLVM – Software Protection for the Masses. In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. 2015, pp. 3–9. Available from DOI: `10.1109/SPRO.2015.10`.

24. *Swiss startup protects SnapChat*. 2017. Available also from: `https://www.startupticker.ch/en/news/july-2017/swiss-startup-protects-snapchat`.

25. *A Simple UPX Malware Technique*. 2020. Available also from: `https://www.mosse-security.com/2020/09/29/upx-malware-evasion-technique.html`.

26. EYROLLES, Ninon. *Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools*. 2017. PhD thesis.

27. MING, Jiang; XU, Dongpeng; WANG, Li; WU, Dinghao. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. [N.d.]. ISBN 9781450338325. Available from DOI: `10.1145/2810103.2813617`.

28. KING, James C. Symbolic Execution and Program Testing. 1976.

29. GUELTON, Serge; GUINET, Adrien. Building, Testing and Debugging a Simple out-of-tree LLVM Pass. 2015. Available also from: `https://llvm.org/devmtg/2015-10/slides/GueltonGuinet-BuildingTestingDebuggingASimpleOutOfTreePass.pdf`.

30. MERLINI, Adrien. *Turning Regular Code Into Atrocities With LLVM*. 2015. Available also from: `https://blog.quarkslab.com/turning-regular-code-into-atrocities-with-llvm.html`.

31. *Build your first LLVM Obfuscator*. 2020. Available also from: `https://polarply.medium.com/build-your-first-llvm-obfuscator-80d16583392b`.

32. *Solving easy CTFs with Angr and Symbolic Execution*. [N.d.]. Available also from: `http://blog.k3170makan.com/2019/12/symbolic-execution-0x0-solving-easy.html`.

33. *Reverse Engineering Snapchat (Part I): Obfuscation Techniques*. [N.d.]. Available also from: `https://hot3eed.github.io/2020/06/18/snap_p1_obfuscations.html`.

A Building and running the passes

Building and testing have been done using LLVM 9. The pass for obfuscating constants (`llvm-pass-obfconst`) requires the C++ `boost` library.

To build the out-of-tree obfuscating passes, simply execute the following shell commands from the root directory:

```
mkdir build
cd build
cmake ..
make
```

Each pass will be compiled into a sub-directory of `/build`. To obfuscate a program, first generate a bitcode module, either in `.ll` or `.bc` format:

```
clang -S -emit-llvm <path to program source>
```

Then, each of the passes can be dynamically loaded with `opt` and used to obfuscate the program:

```
opt -load /build/llvm-pass-<pass-name>/<compiled pass> -<pass flag>
↪ -S <path to bitcode file> -o <output bitcode file>
```

For example, to apply Instructions substitution (MBA) pass, execute the following:

```
opt -load /build/llvm-pass-mba/libMbaPass.so -mba -S foo.ll -o
↪ foo_obfuscated.ll
```

Afterwards, use `clang` (or `clang++`) to obtain the obfuscated binary.

```
clang foo_obfuscated.ll -o foo_obfuscated
```

If the project you want to obfuscate contains multiple source files, you can either obfuscate the bitcode files separately and link them with `llvm-link`, or link them first and obfuscate the resulting bitcode file.

Following flags must be used to apply corresponding passes:

- MBA (Substitution): `-mba`
- Bogus Control Flow: `-bogus`
- Constant obfuscation: `-obfconst`
- String obfuscation: `-obfstring`

To use the String obfuscation pass, a `codec.bc` file needs to be present in the directory from which the `opt` tool is being executed. It can be generated by executing:

```
clang -c -emit-llvm <path to codec source> -o codec.bc
```

Two example implementations of codec are included in the `llvm-pass-obfstring` directory. The source code needs to contain functions named `encode` and `decode` and they need to have a single argument of type `unsigned char *`.