KTH Computer Science
and Communication

Platform Independent Code Obfuscation

OSKAR ARVIDSSON

Master's Thesis at CSC
Supervisor: Mikael Goldmann
Examiner: Johan Håstad

TRITA xxx yyyy-nn

Abstract

Code obfuscation is a technique used to make software more difficult to read and reverse engineer. It is used in the industry to protect proprietary algorithms and to protect the software from unintended use. The current state of the art solutions in the industry depend on specific platform targets. In this report we look at code obfuscation from a platform independent point of view. The result is a survey of code obfuscation methods that can be used together to perform platform independent code obfuscation. We also analyze some of these methods in more detail and provide insights regarding their potency (difficulty to deobfuscate manually), resilience (difficulty to deobfuscate automatically), stealth (difficulty to distinguish from normal code) and ease of integration (how easily the method can be integrated and used in a toolchain).

Referat

Plattformsoberoende kodobfuskering

Kodobfuskering är ett verktyg för att göra mjukvara svårare att läsa, förstå och bakåtkompilera. Det används inom industrin för att skydda proprietära algoritmer samt för att skydda program och tjänster från att missbrukas. De lösningar som finns att tillgå idag är dock ofta beroende av en eller flera specifika plattformar. I den här rapporten undersöker vi möjligheten att göra plattformsoberoende obfuskering. Resultatet är en undersökning av vilka obfuskeringsmetoder som finns tillgängliga, samt en djupare studie av några av dessa. Den djupare studien ger, för var och en av de studerade metoderna, insikter om hur svåra de är att deobfuskera för hand, hur svåra de är att deobfuskera automatiskt, hur pass svårt det är att skilja den obfuskerade koden från den oobfuskerade, samt hur lätt det är att implementera och integrera dem i en kompileringskedja.

Acknowledgements

The work that this report is based on was in part carried out together with Oskar Werkelin Ahlin. I would like to thank him both for his support during the study, implementation and evaluation of different code obfuscation techniques, and for his encouragement during the whole process. Next I would like to thank my supervisors Mikael Goldmann and Henrik Österdahl. They have given me excellent feedback during the course of the project. Without their effort this report would not have become what it is. Finally I want to thank Johan Håstad for his advice and feedback during the final work on the report.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Purpose . . . 2
- 1.3 Outline . . . 3

2 Background **4**
- 2.1 Terminology . . . 4
- 2.2 Code obfuscation . . . 4
  - 2.2.1 History . . . 4
  - 2.2.2 Definition . . . 5
  - 2.2.3 State of the art . . . 5
- 2.3 Compiler technology . . . 6
  - 2.3.1 Front end . . . 7
  - 2.3.2 Back end . . . 9
  - 2.3.3 Common compilers . . . 11
- 2.4 Reverse engineering . . . 13
  - 2.4.1 Static analysis . . . 13
  - 2.4.2 Dynamic analysis . . . 14
  - 2.4.3 Program slicing . . . 15
- 2.5 Code obfuscation methods . . . 15
  - 2.5.1 Layout obfuscation methods . . . 16
  - 2.5.2 Data obfuscation methods . . . 16
  - 2.5.3 Control flow obfuscation methods . . . 19
- 2.6 Obfuscation quality . . . 25
  - 2.6.1 Potency . . . 25
  - 2.6.2 Resilience . . . 27
  - 2.6.3 Execution cost . . . 27
  - 2.6.4 Stealth . . . 28
  - 2.6.5 Ease of integration . . . 28

3 Method **29**

3.1 Assumptions . . . 29
3.2 Obfuscation methods . . . 29
3.2.1 Function pointer exploitations . . . 30
3.2.2 Function transformations . . . 31
3.2.3 Exception exploitations . . . 34
3.2.4 Value encoding . . . 35
3.3 Evaluation . . . 38

4 Implementation **41**
4.1 Tools . . . 41
4.2 Obfuscator . . . 42
4.2.1 Obfuscation methods . . . 42
4.2.2 Constant pool . . . 42
4.2.3 Opaque predicates . . . 42
4.2.4 Toolchain integration . . . 43
4.3 Testing . . . 44
4.4 Analysis . . . 44

5 Results **45**
5.1 Potency and cost . . . 45
5.1.1 Value encoding . . . 46
5.1.2 Function pointer transformations . . . 47
5.1.3 Function transformations . . . 48
5.1.4 Exception exploitation . . . 48
5.1.5 All obfuscations . . . 49
5.2 Resilience . . . 50
5.2.1 Value encoding . . . 50
5.2.2 Function pointer transformations . . . 50
5.2.3 Function transformations . . . 51
5.2.4 Exception exploitation . . . 52
5.3 Ease of integration . . . 52
5.3.1 Value encoding . . . 52
5.3.2 Function pointer transformations . . . 52
5.3.3 Function transformations . . . 53
5.3.4 Exception exploitations . . . 53
5.4 Stealth . . . 53
5.4.1 Value encoding . . . 53
5.4.2 Function pointer transformations . . . 54
5.4.3 Function transformations . . . 54
5.4.4 Exception exploitation . . . 55
5.5 Summary . . . 55

6 Discussion **56**
6.1 Conclusions 56
6.1.1 Theoretical Results 56
6.1.2 Subjective Thoughts 57
6.2 Future work 58

Bibliography **59**

Chapter 1

Introduction

There are many different techniques for a company to protect its intellectual ideas against competitors and their services against unintended use. This report studies one such technique. Code obfuscation is a collection of methods that can be used to protect software against reverse engineering. Reverse engineering is a process that transforms the output of another process into the input of said process in part or completely. An example of this is to reconstruct the source code from a compiled program.

The application of code obfuscation does not guarantee that a program cannot be reverse engineered, but its purpose is to make the process of reverse engineering prohibitively expensive. Combined with other techniques for protecting software against third party tampering, code obfuscation can serve as an important tool for software protection in the industry.

1.1 Motivation

Consider the process of distributing software from one party to another. One party acts as distributor (the company) and the other party acts as user of the software. According to tradition, we will call the first party Alice and the second party Bob. Alice wants to distribute her software service in a way so that Bob cannot easily reverse engineer it and reveal the secrets of her code. The secrets can be anything from encryption keys to algorithms. It is important that Bob can run the software efficiently in the way it was intended to, otherwise Bob might choose to use another service instead.

As of today, there are many ways to approach this problem. Alice can choose to distribute the software as an online service, keeping the important parts of the code away from Bob's computer. This could be done with the entire application, or a limited part of it. Distributing it as an online service

requires that Bob has Internet access while running the program, and it also might require Alice to solve the problem of dealing with high server load.

Another solution for Alice could be to use hardware support to secure the software. A specific hardware device would be required for using the service. This would require an attacker to analyze how such a device works. Even though such an approach seems secure, it would be tedious to distribute such a program due to the requirement of a physical hardware device. Furthermore, if an attacker is able to disable or bypass the device, the system would still be open for attacks.

Code obfuscation aims to hide Alice's secrets through obscuring the code from third party inspection. Code obfuscation does not obstruct Bob from running the program in the way he usually does. In fact it preserves its functionality, albeit at a slight decrease in performance. Bob may try to reverse engineer the code, but the obfuscations will make that process more difficult, time consuming and bothersome.

Let us consider the following scenario: Alice wants to distribute her software to as many customers as possible, so she is planning to do it for desktops as well as mobile clients. Most of the customers use standard operating systems such as Microsoft Windows or Apple Mac OS X on a personal computer. But there are also people who would like to use her software on their smart phones, tablets, etc. Recently she has also been approached by companies who want to embed her application into their hardware. She wants to protect her software on all these platforms.

Over the years, tools have appeared that can obfuscate code for specific platforms, but since they depend on specific platforms, they need to be revamped as new platforms appear. Each of them implements different obfuscation methods. The methods have different characteristics regarding how difficult they are to implement, the cost of using them and how easy they are to remove by an attacker. We are interested in looking at a subset of these methods that are independent of the target platform.

1.2 Purpose

The long-standing problem of protecting software secrets against a potentially malicious third party today needs to be addressed for multiple platforms, ranging from common personal computers to integrated systems with specially designed operating systems. Even more so, with new platforms emerging constantly, platform independent code obfuscation could be a potent, economic and practical protection against a malicious third party.

In this report we summarize the existing methods for code obfuscation and

reason about how they can be used independently of target platform. The most interesting methods are evaluated and analyzed according to common practice in the field. We also analyze and evaluate combinations of them.

Finally, the aim of the report is to present some recommendations on how a system for target platform independent code obfuscation could be built.

1.3 Outline

Chapter 2 serves as an introduction to the current research in the field, both in terms of what methods for code obfuscation exist and how they can be evaluated and analyzed. In chapter 3 we lay out and motivate the selection of the approach that was used for evaluating the chosen obfuscations. We also present and motivate our choice of analysis methods, which have been derived from current research in the subject area. Chapter 4 discuss in more detail how the obfuscation methods were implemented. In chapter 5 we present the results of the analysis. Finally we present our conclusions in chapter 6, where we also discuss future work and what could have been done differently.

Chapter 2

Background

In this chapter we describe important theory and concepts that can aid in understanding the following sections.

2.1 Terminology

This section define terms that are used throughout the report.

Obfuscator A program that obfuscates an input program, emitting an obfuscated output program with equivalent semantics compared to the input.

Toolchain A set of tools that are used to create a software product. Tools are usually chained in a specific way. A simple example of a toolchain is a source code editor and a compiler used to compile the source code.

CPU Central processing unit, mostly referred to as the computer processor.

2.2 Code obfuscation

Code obfuscation is a collective term for techniques that make code more difficult to read and reverse engineer.

2.2.1 History

Techniques for code obfuscation were first formalized by Collberg et al. 1997 [4]. Although code obfuscation had been used prior to this date, academic research in the area had been sparse until then. At first there were hopes that obfuscation could be used as a black box providing encryption similar to public key encryption. In 2001 Barak et al. [2] showed that no obfuscation

could exist in this sense. The results found by Barak et al. more or less halted the academic research in the area of code obfuscation for many years, although companies kept using it practically. Little academic progress was made until 2004 when Lynn et al. [21] showed the first positive results about obfuscation. They showed how obfuscation can be applied on access control graphs, and observed that a similar approach probably could be used for obfuscating finite automata or regular expressions.

Code obfuscation has been used in one way or another in the industry for a long time. Lately, its usefulness in malware creation has been discovered [26], making it more difficult to do code analysis on viruses thus making it more difficult for anti-virus programs to detect malicious software. Much of the academic research during the last years has thus focused on deobfuscating obfuscated software in order to protect users from malware.

2.2.2 Definition

In order to reason about the correctness of an obfuscation method, we first need to define formally what an obfuscation is. Collberg et al. [4] proposed the following definition:

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

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

Collberg et al. defined the observable behavior to be the behavior of the program "as experienced by the user". What this means is that P' may have side-effects that P does not have, but they should be acceptable and go unnoticed by the user. Collberg et al. further elaborates about the memory usage and speed difference between P and P' and states that such differences are valid side effects.

2.2.3 State of the art

There are a few proprietary obfuscation tools available for usage. Most of them are designed for a specific platform in order to allow for increased efficiency, requiring software running on different architectures to use different obfuscation tools.

Themida

Themida is a proprietary software protection system for Windows developed by Oreans Technologies. It works on a binary level, obfuscating compiled Windows x86 binaries. Among its notable features it includes anti-debugging, anti-memory dump (a snapshot of the working memory of a process at a specific time) and binary integrity checks (ensures that the program can only run if not modified). It also includes functionality for hiding code inside a virtual machine which is obfuscated in itself, increasing the obfuscation level significantly. As Themida not only obfuscates code, it is much more than an obfuscation tool, targeting to protect a program from reverse engineering altogether. [28]

Morpher

Morpher is a compiler driven obfuscation tool developed by MTC Group LTD. Morpher has support for a large number of obfuscation methods, and supports using them in arbritary combinations. It makes use of its tight coupling with the languages' compilers to be able to apply more sophisticated transforms which require more information about the source code.

It has support for standard C/C++, and limited support for Fortran95 and Ada. Supported architectures are x86, PowerPC and ARM among others. The obfuscator tool is built on llvm-gcc (see Section 2.3.3). Among its notable features Morpher has support for protecting constant values (e.g. values are stored encrypted, and decrypted only upon use) and function transformations. [20]

Diablo

Diablo is an open source infrastructure for rewriting binaries. Development is currently stalled, but some obfuscation methods have been implemented previously. One of the features is control flow graph flattening. Support exists for x86 and ARM. [14]

2.3 Compiler technology

A compiler is by definition a computer program that transforms source code into another form, e.g. native code. The process of compiling source code can usually be divided into a number of steps, which are briefly described in this section. A standard compiler can be divided into two main parts, namely the front end and the back end. There are no exact rules for how a compiler

should be written; the description below rather describes one approach. This section is based on *Engineering a Compiler* by Cooper and Torczon [6].

2.3.1 Front end

Part of the compiler front end is lexical analysis, parsing and semantic analysis. The lexical analysis and parsing are often jointly referred to as syntactical analysis.

Lexical analysis

Lexical analysis is the process of transforming the flow of characters of the input source code into a sequence of tokens describing the program. Conversely, a token describes a sequence of characters in the source code. Examples of tokens that can be useful for a C compiler front end are **IF**, **ELSE**, **FOR**, **RETURN** and **AND**, but a C string literal such as "a string" can also make up a token. The purpose of the lexical analysis is primarily to make the code easier to process for the parser. Listing 2.1 shows a program and Listing 2.2 shows a possible result from an imaginary lexer when fed with the program in Listing 2.1. Note that each token in the lexical analysis often is associated with the sequence of characters it represents in the source input. This information is needed to e.g. extract the contents of a string or the name of an identifier.

```
1 int main(int argc, const char **argv) {
2         printf("%d %s\n", argc-1, argv[0]);
3         return 0;
4 }
```

Listing 2.1. A simple example program.

```
1 INT ID PAR_BEGIN INT ID COMMA CONST CHAR STAR STAR ID PAR_END
    CURLY_BEGIN ID PAR_BEGIN STRING COMMA ID MINUS NUM COMMA ID
    SQUARE_BEGIN NUM SQUARE_END PAR_END SEMICOLON RETURN NUM
    SEMICOLON CURLY_END
```

Listing 2.2. Example of what the lexical analysis of the program in Listing 2.1 might look like.

Parsing

Input to the parser is the output from the lexical analysis. Output from the parser is an abstract syntax tree (AST). Nodes in the AST are abstract concepts of the source language. A node may have static attributes, such as the name of the function in a function declaration, and child nodes, such as the arguments in a function call. The parser ensures that the input is part of the source language grammar, and emits a syntactic error if this is not the case. Figure 2.1 shows the output from an imaginary parser fed with the output from Listing 2.2.

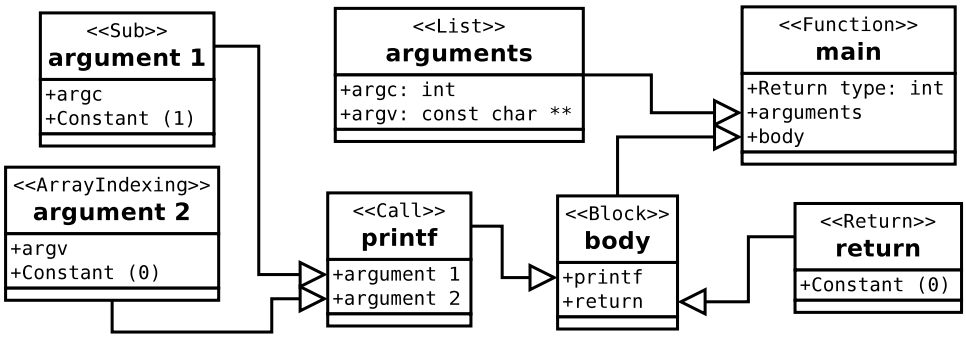

Figure 2.1. Example of an AST that could be generated from the program in Listing 2.1.

Semantic Analysis

Input to the semantic analysis is an AST. The purpose of the semantic analysis is solely to ensure that the program is semantically correct. Common errors that the semantic analysis should detect are references to non-existing identifiers (names of variables, functions and similar constructs), type errors, etc. To help in the process, the compiler creates a symbol table, which contains information about the visible identifiers in different scopes within the code. Apart from being used as a means to do semantic analysis of the code, the symbol table can be output for further use by later phases in the compiler.

Intermediate code generation

A compiler may support multiple input languages and multiple output formats. Common input languages are for example C and C++, and it is not uncommon that a user of a compiler will want to target two or more completely different platforms, such as x86 and ARM (we ignore other platform

differences such as operating system here). One way to support this is to write one compiler for compiling C code to x86, another for C++ to x86 and similarly one more compiler able to compile source of each input language to ARM native code. In the general case this would require $M \times N$ compilers, where M is the number of input languages and N is the number of output formats.

While specialized compilers may sometimes be useful, a more general-purpose approach is to write one front end for each input language that compiles the code to a common intermediate language, and one back end for each output format. Figure 2.2 illustrates this process. The intermediate representation needs to be complex enough to cover all constructs in all input languages. In contrast to the specialized approach, with $M \times N$ compilers, this approach only requires M compiler front ends and N compiler back ends. Furthermore, the most difficult and advanced part of a compiler is analysis and optimizations. Parts of the analysis and optimizations can be done on the intermediate code, reducing the amount of work required to write a back end for a new output format.

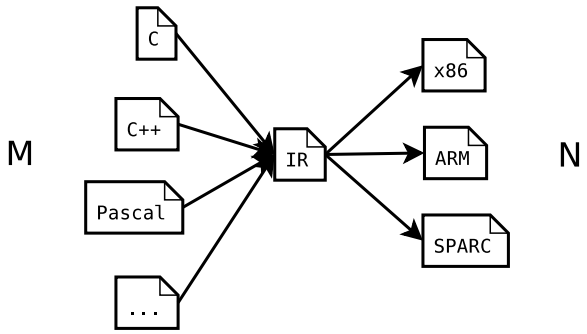

Figure 2.2. An illustration of the purpose of the intermediate code.

It should be noted that almost every compiler uses its own language and conventions for intermediate code (IR). One common property shared by many of these languages is single static assignment (SSA). In SSA each variable is assigned exactly once, i.e. it is not allowed to write to a distinct variable multiple times. A simple example of IR code on the SSA form can be seen in Figure 2.3.

2.3.2 Back end

The compiler back end is responsible for analysis, optimization and code generation. The code generation step transforms the program into object code,

$$
\begin{array}{ll}
x \leftarrow 0 & x_1 \leftarrow 0 \\
y \leftarrow 5 & y_1 \leftarrow 5 \\
 & \\
y \leftarrow x * y & y_2 \leftarrow x_1 * y_1 \\
x \leftarrow x * y & x_2 \leftarrow x_1 * y_2
\end{array}
$$

Figure 2.3. A sample program to the left and its SSA transformation to the right.

which contains compiled code and some associated information such as exported symbols, for the target machine. A file containing object code is called an object file. Finally a linker phase is needed in order to combine multiple object files into one executable file.

Analysis and optimizations

One of the most important properties of a compiler is the quality of its optimizations. In order to do optimizations on the code, the compiler needs to analyze the code. The analysis done by a compiler has much in common with static analysis that an attacker can perform as described later.

Input to the optimization phase is intermediate code. The intermediate code is parsed into one or more control flow graphs (CFG) which are similar to an AST, but on a much more detailed level. The nodes in the CFG consist of basic blocks. One basic block is a small piece of code that fulfills certain requirements. Most importantly, only the last instruction in a basic block can be a branch instruction. Each branch instruction corresponds to a directed edge in the CFG from one basic block to one or more basic blocks. In the most basic case, each function in the source code will translate to one CFG. A CFG may have one or more exit nodes, corresponding to different return points in a function. A CFG normally has one entry point, corresponding to the start of the function it models. Note that a CFG may also be used to model things apart from functions such as whole programs. Figure 2.4 shows a control flow graph based on the sample code in listing 2.3.

The purpose of control flow graphs is to simplify the process of performing analysis and optimizations. The control flow graphs can also be used for optimization purposes itself. For example, basic blocks with no incoming edges can be discarded as they contain unreachable code.

```
1 void f(int x) {
2     ++x; // block 1
3     if (x == 1)
4         x = 0; // block 2
5     else
6         ++x; // block 3
7     printf("%d",x); // block 4
8     return;
9 }
```

Listing 2.3. A sample program.

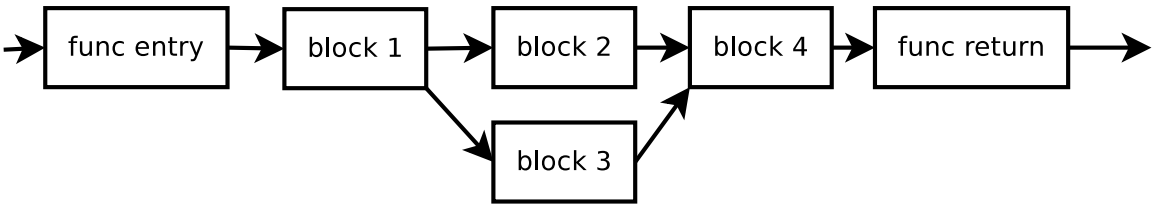

Figure 2.4. The control flow graph of the function in Listing 2.3. Nodes represent basic blocks.

Code generation

Input to the code generation is intermediate code. The code generation phase of the compiler is responsible for instruction selection and register allocation. Instruction selection transforms the instructions of the intermediate code to instructions in the target's native code such that the semantics of the program is preserved. Register allocation is the process of assigning processor registers efficiently to the variables that need to be represented in the program. When required, the register allocation will spill registers to the stack. Note that instruction selection as well as register allocation is subject to optimizations as well. Output of the code generation phase is native code for the target platform.

2.3.3 Common compilers

Here we present a survey of some common compilers for C and C++.

GNU Compiler Collection

The GNU Compiler Collection (GCC) is an open-source compiler. It is produced by the GNU project, a free software and mass collaboration project. It has been adopted to be the standard compiler by most Unix-like operating systems, but support exists for Windows as well. GCC uses an intermediate representation that is called GIMPLE. GIMPLE comes in many forms, of which one is SSA based.

GCC is widely used not only in free software projects, but also in commercial and proprietary software development. [11]

Low Level Virtual Machine

Low Level Virtual Machine (LLVM) is an open source compiler infrastructure which originally was a research project at the University of Illinois. LLVM is commonly used together with Clang which is a compiler front end with support for C/C++/Objective-C. In this configuration, LLVM acts as the compiler back end. There is also a GCC compatible front end for LLVM called llvm-gcc. This front end intends to be a drop in replacement for GCC on supported platforms, while using the LLVM back end.

One of the main features of LLVM is that it is modular, making it relatively easy to extend. In particular the LLVM documentation covers writing modules that analyse or transform the code. Many of the optimizations done by LLVM are implemented as modules. This architecture stands in contrast to other compilers, where transformation code needs to be built statically into the compiler. Another feature of LLVM is the LLVM IR. LLVM IR is in SSA form, and the same IR can be used as input to all LLVM tools, such as the optimizer, the code generator and interpreter. It can also easily be edited manually without any tools other than a text editor.

LLVM is available for multiple processor architectures, and is able to produce code for x86, x86-64 and ARM as of today. Support for other architectures can be added with modules just as with optimization passes. [19]

Microsoft Visual Studio Compiler

The Microsoft Visual Studio Compiler, which is distributed as part of Microsoft Visual Studio is the de facto compiler for Windows programs – although many other compilers exist. The Microsoft Visual Studio Compiler is proprietary, and can not be extended with third party code transformation modules. [7]

2.4 Reverse engineering

In this section, different reverse engineering techniques are explained and discussed. We do not go into great depth, since this is not the topic of the report. It is necessary to have a brief understanding of some standard reverse engineering techniques in order to understand the benefits of certain obfuscation techniques.

Analysis methods are divided into two main types, namely static and dynamic analysis.

2.4.1 Static analysis

Static analysis refers to analysis that is carried out without executing a program. The analysis is carried out on source code describing the program. Generally the original source of a program is not available for static analysis, but only an executable without debugging symbols. Thus static analysis depends on a correct transform from the executable code to a source code representation of the program. This transformation is normally done with a disassembler. Disassembly is not an exact process. In fact, disassembly in general is known to be reducible to the halting problem, thus unsolvable. This needs to be noted when working with static disassembly. Although the static analysis may be correct for the disassembly at hand, the disassembly in itself need not be correct thus invalidating the static analysis. [25]

Data-flow analysis

Data-flow analysis is a technique for determining the set of possible values for variables in certain points of a computer program. For each basic block, we define an entry state s_{entry} and an exit state s_{exit}. By state, we mean information about the program, e.g. relationships between variables. Define a transfer function f such that $f(s_{entry}) = s_{exit}$. We also know that s_{entry} depends on the combined exit states of predecessors of s in the control flow graph. Using this, we get the pair of equations for each node:

$$s_{entry} = \cup_{s' \in S} s'_{exit}$$
$$s_{exit} = f(s_{entry})$$

where S is the set of predecessors to s. By solving these equations, we can determine certain properties about the data flow in each node. [6]

Code cloning

In a program, the same node can often be reached through different execution paths. As the information propagated by data-flow analysis will be more complex for the node in this case, a technique called code cloning is commonly used. Code cloning aims to duplicate nodes so that each node is reachable through exactly one path of nodes. For example, if block C is reachable either through block B or block A, C is split into two blocks C_A and C_B. This way, each block will have exactly one predecessor, making the control flow graph larger, but also more simple. [29]

Path feasibility analysis

Path feasibility analysis finds a subset of the dummy edges introduced by control flow obfuscation, and concludes that they are unfeasible, giving the reverse engineer a more simple control flow graph.

Assume that we have an arbitrary acyclic program execution path P, and $\bar{x}$, the set of live variables (variables in use) at entry to P. We want to construct a constraint C_P such that $(\exists \bar{x})C_P$ is unsatisfiable only if P is unfeasible. Having constructed C_P, we check if it is satisfiable. If it is not, we know that P is unfeasible.

The condition C_P can be constructed using a simple set of rules that depend on the operations performed along the execution path, as explained by Udupa [29].

2.4.2 Dynamic analysis

Dynamic analysis is carried out during program execution. The software is executed on a real or a virtual processor. As opposed to static analysis, which should always give the same result for a specific input, the result of dynamic analysis is highly dependant on how the program is executed. Therefore, it is important to explore a sufficient amount of different program executions to create interesting program behaviour. Techniques such as code coverage can be used to ensure that a sufficient portion of the program's set of possible execution paths has been explored. During the run, various data can be collected. Examples of such data can be snapshots of the state of the program at different locations in the code, and the control flow graph as traversed when executing the program. [29]

2.4.3 Program slicing

Program slicing is an important technique that can be used both for static and dynamic analysis. A program is sliced according to some slicing criterion, and then all parts that are not affected by the parts of interest are filtered out. This makes the debugging process easier, for example if we want to know why a specific value is incorrect at a specific point in the program, the variable would be selected and slicing would filter out all parts of code that do not affect the variable at hand, directly or indirectly.

```
1 a = input()
2 c = 5
3 b = a
4
5 print b
```

```
1 a = input()
2
3 b = a
4
5 print b
```

Figure 2.5. A sample program to the left and to the right the result of program slicing on the last statement.

Consider for example the code in Figure 2.5. Assume that the statement `print b` made the program output an unexpected statement. We would slice the program according to this statement, and the slicer would filter out all values that are independent of it. Looking at the example above, slicing on the variable a would filter out the statement `c = 5` as it does not affect the variable a in any way. [16]

2.5 Code obfuscation methods

The methods that can be used to obscure source code can be divided into three categories; layout obfuscations, data obfuscations and control flow obfuscations. Layout obfuscation only deals with the syntactic elements of the source code, i.e. how the source code is formatted and encoded. Data obfuscations obscure variables, classes and other data. Control flow obfuscations change the control flow of the program such that its pattern is less obvious to spot while analyzing the program flow. This section is largely based on the work by Collberg et al. [4] and Drape [9]. Note that we put the emphasis on describing the concept of the methods, rather than elaborate on when they can be used without changing the semantics.

2.5.1 Layout obfuscation methods

Layout obfuscations have in common that they only change syntactic elements in the code, i.e. they change the appearance of the code while leaving the real structure intact. Normally this type of transformation is unnecessary as the compiler already removes this structure from the code. However if the code is to be redistributed as is, i.e. in source code, methods of this type can be used to remove some of the human readable information. For source distribution, these transformations actually can provide an important means of protection as variable names and other syntactic sugar provide a human reader with context for better comprehension of the code even if it have been obfuscated by other means.

Scrambling identifiers

Scrambling identifiers is the process of renaming identifiers such as variable and function names. It aims to change these identifiers to names which do not explain what they are used for, or to names that are illogical to what they are used for. Figure 2.6 shows an example of this.

```
int confirmLogin() ...          int apples() ...
```

Figure 2.6. An example of how identifier names can be rewritten.

With most compiled languages, this information is not preserved by the compiler – if the identifiers are not explicitly or implicitly exported in any way such as through the use of debugging information. Removing this type of information from native executables is called *stripping*.

Remove comments

Comments often contain high level information such as why the code works in a specific way, and what the purpose of the code is. This informatin can be removed without any semantic changes. Similarily to the process of scrambling identifiers, this process is normally only useful if the code is disitributed in source form as a compiler does not preserve this information.

2.5.2 Data obfuscation methods

A data obfuscation method changes the way that data is stored in memory. Instead of storing a data structure in the normal way, data is shuffled or

changed so that it is difficult to interpret at run time, without knowing in which way its representation has been changed.

Value encoding

Encryption of constant strings is an example of value encoding obfuscation. The idea behind this obfuscation method is that an attacker will inspect variables during execution in order to understand the context in which the variable is used. Incrementation of a variable by 1 in a block of code with the variable compared to a constant limit is what a typical loop would look like. Similarly branching based on a string value is easy to spot and can help the attacker navigate in the code.

An example of value encoding obfuscation is to encrypt each constant string in the code. Upon usage the string is decrypted and after use it is encrypted again. It is possible to perform a similar obfuscation for integers. Instead of coding a value naturally, it can be coded in a way similar to Equation 2.1 ($\oplus$ means exclusive or in this context).

$$var' = var \oplus 17 \tag{2.1}$$

A desirable feature for a value encoding obfuscation is that it is a one to one mapping over the range of the variable that stores the value. This ensures that the obfuscation is invertible, i.e. whatever the value of the variable to be encoded is, the value can be decoded without ambiguity. Hence the obfuscation will not break the program if the range of the variable is changed without the obfuscators knowledge.

Variable aliasing

Variable aliasing works by changing the way in which variables are stored, e.g. by splitting them up into several variables or merging them into a single variable. A simple example is the merging of two equally sized integers into one integer with the double size. This can be done by simply storing the first integer in the lower part of the new integer and the second integer in the upper part. Storing the variables in this way generally requires the new variable to be repacked for each operation that operates on either of the original variables. Listing 2.4 shows an example of how variables can be aliased.

Class refactoring

Class refactoring obfuscates by changing the class inheritance hierarchy. This can be done through for example inheritance from dynamically generated

```
int8_t a = ...;
int8_t b = ...;
int8_t c = ...;
int8_t d = ...;

int32_t merged = a | (b << 8) | (c << 16) | (d < 24);
```

Listing 2.4. An example of how four variables can be merged into one.

classes with no real functionality, or by taking all functionality of one class and split it into two new classes while the original class acts as a proxy for the two new classes. Figure 2.7 shows one such example.

```
class InputCheck {
    void updateInput();
    int getResult();
};
```

```
class A {
    void updateInput();
}

class B {
    int getResult();
};

class InputCheck :
    public A, B {}
```

Figure 2.7. Example showing how the code to the left can be transformed according to the class refactoring obfuscation method. Note that information cannot be shared between class A and B. In case that is needed, diamond inheritance might be required.

Array restructuring

Array restructuring is an obfuscation method that changes the structure of an array, transforming it in a way that makes it difficult for an attacker to understand its structure at run time. A simple example of array restructuring would be reversing it at compile time, and reading it in the opposite order at run time effectively undoing the reversal done during compilation. Listing 2.5 shows an example of array restructuring.

```
int scan[10] = { 8, 2, 9, 0, 6, 4, 1, 3, 5, 7 };
int fibonacci[10] = { 2, 8, 1, 13, 5, 21, 3, 34, 0, 1 };

for (int i = 0; i < 10; ++i)
  printf("%d\n", fibonacci[scan[i]]);
```

Listing 2.5. An example of how an array can be restructured for obfuscation purposes.

Variable promotion

The motivation behind variable promotion is to make it more confusing which context a variable belongs to by increasing the scope of said variable. For example a loop variable is commonly initialized just before the loop, used for array indexing inside the loop and at the end of the loop incremented or decremented. After the loop it is usually not used for anything. Instead of this scheme, an obfuscator can initialize the loop variable implicitly in some other context, use the loop variable in the loop, and then continue to use it outside the loop context, as shown in Listing 2.6.

```
int i;
for (i = 0; i < 10; ++i)
    check(i);

char buffer[i];
scanf("%9s", buffer);
```

Listing 2.6. An example of how a variable can be promoted for obfuscation purposes.

2.5.3 Control flow obfuscation methods

Control flow obfuscations aim to complicate the control flow graph of a program. They generally do this by adding more program states and branches. This makes the debugging process of the program more tedious. Some control flow transformations are similar to the transformations that the compiler performs on the code when optimizing it. Some common optimizations may be reverse transformations to control flow obfuscations. Thus clever optimizations may undo control flow obfuscations.

Opaque predicates

Opaque predicates are predicates which will evaluate to a, at compile time, known value. Using this fact, we can create conditionals which will complicate the control flow graph. This could be done by e.g. using a mathematical identity as in Listing 2.7 [5].

```
1 int v = rand();
2 if ((v * v * (v+1) * (v+1)) % 4 == 0)
3     // always executed
4 else
5     // never executed
```

Listing 2.7. An example of dynamically created opaque predicates.

An obfuscator could also use relations between variables acquired through e.g. statical analysis to create opaque predicates. The obfuscator could also choose to create such relations itself through the introduction of one or more new variables with a predecided relation.

Opaque predicates are commonly used as part of other obfuscation methods to produce more powerful obfuscations.

Pseudo cycles insertion

Pseudo cycles insertion creates a loop in the code with some kind of opaque predicate as loop condition, which makes the program always break out of the loop immediately, but for a static analyzer it will look like there is another loop in the program. This further complicates the control flow graph of the program.

Control flow flattening

The control flow graph visualizes the program flow and can thus help an attacker in understanding the program structure. The control flow can also help to trace at what places in the code a specific function is called. Control flow flattening is a collection of methods for obscuring this structure by reducing the height of the control flow graph. In the extreme case, all jumps between basic blocks in the program are replaced by jumps to a common proxy which relays each jump to its original jump destination. This method effectively reduces the control flow graph to two levels – one level containing all original basic blocks, and one level with the proxy, with edges between the proxy and each basic block entry point and edges between each basic block's exit point and the proxy entry point, as shown in Figure 2.8.

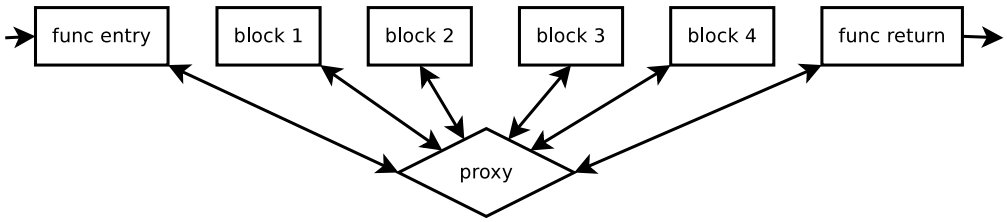

Figure 2.8. A program where all jumps between basic blocks go through a common proxy block.

Combined with opaque predicates, this obfuscation method can make it much more difficult for the attacker to determine what path will be executed just through static analysis.

Function pointer obfuscation

All common programming languages have support for function calls. Some programming languages also have support for function pointers. In contrast to normal functions, it may be difficult to determine what function is actually called when a function pointer is used instead.

One approach to function pointer obfuscation is to store a pointer to each function in a globally accessible array. Each function call is then replaced by indexing into this array, loading the pointer that corresponds to the function to call and finally calling that function pointer. This scheme can be extended with other obfuscations, such as value encoding and variable aliasing, effectively obscuring the function pointer data structure. Combined with opaque predicates, the process of determining statically what function is called can be made even more difficult.

Transforming the code to use function pointers instead of normal function calls have been formally proved to obscure the program. In the general case, determining which function a function pointer call corresponds to has been proven to be NP-hard [23].

Figure 2.9 shows an example of function pointer obfuscation.

Loop transformations

The optimizer in a compiler often performs various loop transformations, for example loop unrolling. Such transformations can also be used by an obfuscator to make the code more difficult to understand. Other obfuscations include changing the loop body into a semantically equivalent but more obfuscated version.

```c
int one_more(int v) {
    return v + 1;
}

int main() {
    int x = 5;
    int y = one_more(x);
    printf("%d\n", y);
    return 0;
}
```

```c
int one_more(int v) {
    return v + 1;
}

int (*func)(int) = &
    one_more;

int main() {
    int x = 5;
    int y = func(x);
    printf("%d\n", y);
    return 0;
}
```

Figure 2.9. An example showing how the code to the left can be obfuscated with the help of function pointers.

Exceptional branching

Common practice is to use some pattern resembling the if-then-else construct when handling conditional control flow in the application. However, exceptions can be used for this purpose as well if they are supported. In fact, an if-else construction can be trivially transformed into a semantically equivalent try-catch block.

This type of obfuscation does not really alter the control flow in the most strict meaning, but it alters the construct that determines the control flow. The motivation behind this is mainly that it can be used to obscure a code block so that it looks like normal exception handling code while it in fact is not, thus misleading a reverse engineer.

Figure 2.10 shows an example of an imaginary exceptional branching transformation.

Dead code insertion

Dead code is code that is either unreachable during program execution or code that does not do anything useful. The motivation behind adding dead code to the program is that it makes the code less clear. A reverse engineer would have to determine what code is executed before he can make any conclusions about what the code does and how it works. Consequently one of the most important attributes when it comes to dead code insertion is that the code is stealthy, i.e. that it does not stand out to the surrounding code. In the case of

```
int check(int input) {
  if (input > 400 && input
      < 800)
    return 1;
  else {
    ...
    return 0;
  }
}
```

```
int check(int input) {
  try {
    if (input > 400 &&
        input < 800)
      throw 1;
    else {
      ...
      return 0;
    }
  } catch (int e) {
    return 1;
  }
}
```

Figure 2.10. A normal conditional control flow to the left, and code that uses exceptional branches to the right.

insertion of unreachable code, there is a need for stealthy opaque predicates to protect the code. Figure 2.11 shows unreachable code and useless code.

```
if (/* false opaque
    predicate */) {
  ...
}
```

```
int check(int v) {
  int p = 0;
  for (int i = 0; i < v;
       ++i) {
    p += i;
    /* code which does not
       reference p */
  }
  return 0;
}
```

Figure 2.11. Unreachable code to the left, and useless code (the code that operates on p) to the right.

Dead code can, like opaque predicates, be useful for other obfuscation methods. One such is exceptional branching, where dead code can be placed in unreachable catch blocks for example.

Function transformations

Function transformations include inline expansion of functions into the caller, splitting one function into two or more functions such that calling the resulting

functions sequentially performs the same action as the original function, and transforming blocks of code into functions.

Functions provide a means for an attacker to navigate in a program. If an important function has been discovered by an attacker he can use this function as a starting-point and track down all the callers of this function. Inlining a function makes it more difficult to do such tracking. Splitting a function into multiple functions makes it more difficult for an attacker to grasp the context of the function. Combined with replacing the split functions with pointers as explained in Section 2.5.3, this can be a powerful obfuscation.

Code virtualization

Code virtualization is an obfuscation method in which the code is transformed into virtual machine code. The code is then executed through the use of a virtual machine interpreter that is shipped along with the program. Code virtualization can be applied on the whole program or only on parts of it.

This method is primarily useful for making the program more time consuming to reverse engineer. To reverse engineer the obfuscated code an attacker may have to reverse engineer the virtual machine used to run the code. The structure of the virtual machine may be very complex, making this a difficult and tedious task. Running the code in a virtual machine interpreter is typically very slow compared to executing the native code directly, thus this approach may not be suitable for performance critical code.

Encryption

Encryption can be applied on different levels in a program. One approach is to encrypt the program in full, and decrypt it in full upon execution. An attacker can in this case either decrypt the program statically or dump the program just after the decryption in run time. In most cases, only parts of the program need to be protected. In this case the full program need not be encrypted, but only the relevant parts.

Another interesting approach to encrypting a program is to do context dependent encryption. Consider a code block protected by a conditional comparing a variable to a constant string for equality. If the code block is executed, the variable will definitely have the value of the constant string. Thus we can encrypt the code block with the constant string as key. To make it more difficult for a reverse engineer to do static decryption, the comparison between the variable and the constant string can be replaced by a comparison between the hash of each value. We know that the hash of the string is not enough to decode the block, hence an attacker would have to run the code to be able

to deduce the meaning of the code in question. Listing 2.8 shows an example of such an approach. Note that this in general requires support from the environment for running self modifying code.

```
1 if (hash(input) == hash("encryption-key")) {
2     /* Decrypt this block with input as key */
3 }
```

Listing 2.8. An example where part of the program has been encrypted based on a string key. The hashing of the string should be done at compile-time for the best result.

2.6 Obfuscation quality

Collberg et al. [4, 5] propose a number of criteria to use when evaluating obfuscation transformations. In particular they define the criteria potency, resilience, execution cost and stealth. Drape [9] added more criteria, of which we were particularly interested in what he defined as ease of integration.

2.6.1 Potency

Potency is defined to measure how obscure a program P is, i.e. how difficult it is for a human reader to understand. It is based on results in software complexity metrics research. Defined are a number of attributes E_i, which are chosen carefully. The program P is defined to be more potent than the program P' with regards to the attribute e if $e(P)/e(P') > 1$. Attributes for measuring potency include cyclomatic complexity, program length and nesting level complexity. Collberg proposed that a weighted sum $E = \Sigma k_i \cdot E_i(P)$ could be used to retrieve one potency value from multiple metrics.

Cyclomatic complexity

McCabe defined a measure called the cyclomatic complexity number [22]. The purpose of the measure is that when a function is more complex, it will have a higher cyclomatic complexity number.

McCabe showed that the cyclomatic complexity C of a function f can be calculated as

$$C(f) = e_f - n_f + 2 \tag{2.2}$$

where e_f and n_f denotes the number of edges and nodes respectively in the control flow graph of the function f. This formula simplifies to

$$C(f) = d_f + 1 \tag{2.3}$$

where d_f denotes the number of conditions in the function f.

Applied to a program, the cyclomatic complexity can be calculated by taking the sum of the cyclomatic complexities of all its functions. The argument for this is that a call to a function is just an edge in the control flow graph of a program.

Halstead's metrics

Halstead defined a number of metrics that can be used for measuring the complexity of a program. For the potency criteria we are interested in one of them, namely Halstead's difficulty metric. Consider a subset of a program, for example one block in a control flow graph, or a function. Let n_1 be the number of distinct operators in the subset. Furthermore let n_2 be the number of distinct operands and N_2 the total number of operands in the subset. Halstead proposed a difficulty metric D for the subset defined as in Equation 2.4.

$$D = \frac{n_1 N_2}{2n_2} \tag{2.4}$$

Halstead claimed that D is positively correlated with the complexity of the subset it is calculated for, and that it can be used to compare different blocks of code with each other.

Nesting level complexity

Harrison et al. [15] argued that neither McCabe's nor Halstead's metrics handled the complexity added by deep nestling of blocks of code correctly. He suggested that the complexity of a control flow graph block should not only depend on what operands and operators that are contained in that particular block, but also on the complexity of the blocks that the block in question can reach. Harrison does not explicitly specify how his new analysis metric should be carried out.

A more precise definition of a complexity measure based on the same idea is proposed by Gong et al. [12].

Assume that we have a directed control flow graph G of a function, with exactly one entry node and exactly one exit node. Let $\mathrm{suc}(x)$ denote the set of immediate successors of x. If and only if $|suc(x)| > 1$, we call x a *selection node*.

Define *postdomination* as follows. For two nodes $x, y \in G, x \neq y$, x postdominates y if and only if every path from y to the exit node passes through x. We say that x *directly postdominates* y if and only if:

- x postdominates y

- $\forall$ z such that z postdominates y, z postdominates x.

We note that trivially all nodes have exactly one direct postdominator when there is only one exit node. For a selection node x, we let G_x contain all the nodes between x and the node that postdominates it. Let d_n be the number of selection nodes in G_n.

Now, we are ready to define our nesting level complexity measure. The nesting degree of a selection node x is defined as in Equation 2.5.

$$\epsilon_n = 1 - (1/d_n) \tag{2.5}$$

The nesting degree of the entire graph is calculated as $\epsilon = (\epsilon_1 + \epsilon_2 + ... + \epsilon_N)/N$, where N is the number of selection nodes in the entire graph. This will yield a nesting complexity measure between 0 and 1, where higher value means higher complexity.

It is difficult to apply the nesting complexity algorithm on a program (i.e. with function calls as normal edges), because of many reasons (indirect function calls, external function calls, etc). It could be argued however that just as for cyclomatic complexity it is possible to get a good result by simply taking the sum of the nesting complexity of all functions in the program.

2.6.2 Resilience

Resilience is a criteria measuring how difficult it would be to create and execute an automatic deobfuscator for a transformation, reversing the obfuscation performed. In contrast to potency this measures the confusion for an automatic deobfuscator, whereas potency measures the confusion for a human deobfuscator. Evaluation of this criteria is quite subjective and speculative, but may be backed up with formal results.

2.6.3 Execution cost

Execution cost is the penalty in terms of speed and memory that an obfuscation imposes. Collberg et al. [4] defined 4 levels for this criteria; free, cheap, costly and dear. An obfuscation is regarded as free if the execution cost is constant, i.e. the penalty is the same regardless of what input the program is run on. It is regarded as cheap if the amount of resources needed to run the obfuscated program is linearly dependent. Define n to be the resources used by the original program, i.e. time and/or memory used. Costly is defined to be any obfuscation that requires $O(n^x), x > 1$, more resources than the original program and a dear obfuscation requires exponentially more resources.

2.6.4 Stealth

Stealth measures how easily a programmer can determine if and how a particular code has been obfuscated. The motivation behind this criteria is twofold. First, an attacker will probably be more interested in a piece of code that looks obfuscated as it probably has been obfuscated for a reason. Second, if the reverse engineer can detect what methods were used to obfuscate a program, he can more easily develop inverse transformations to recreate the original code as well as understand the code easier.

2.6.5 Ease of integration

Ease of integration measures how easily an obfuscation can be applied to an existing toolchain. It takes into account how difficult it is to produce an obfuscation tool for the given method, adapt it to the source at hand and include it in an existing toolchain.

Chapter 3

Method

In this chapter we discuss the method that we will use for collecting the results. We first state some assumptions about the context of the code obfuscation, then we discuss the implementation of the obfuscation methods in more detail and finally we describe how the evaluation of the code obfuscation methods will be performed.

3.1 Assumptions

The programming language has been assumed to be something similar to C or C++, i.e. statically typed and imperative. It is still possible though that some of the topics discussed are applicable for other programming languages.

3.2 Obfuscation methods

In order to complete the project on time, only a few obfuscation methods could be analyzed in detail. After a brief review regarding the ease of integration, potency and resilience, it was decided that the project should focus on four obfuscation methods, namely:

- Function pointer exploitations.
- Function transformations.
- Exception exploitaions.
- Value encoding.

3.2.1 Function pointer exploitations

Programming languages often have support for direct function calls as well as indirect ones. In particual, C [17], C++ [27], Python [10] and Java [13] support indirect function calls, although Java requires the use of classes to implement it.

The primary reason why indirect calls provide more obfuscation than direct calls is that indirect calls are typically dependent on runtime information. Thus it is potentially much more difficult for a static analyzer to determine which function actually gets called from a particular call site. Wang et al. [31] showed that in the general case, the problem of precisely determining indirect branch addresses is NP-hard. In a more specific case the problem is much simpler though. Consider for example the sample code in Listing 3.1.

```
1 int (*function)(void) = other_function;
2 function();
```

Listing 3.1. An indirect call.

Statically determining the exact function that is called in runtime is in this case trivial. An obfuscation scheme that transforms direct calls to indirect calls must thus be more advanced and use more complex strategies.

Wang et al. [31] propose an algorithm for this transformation which uses a global array where function pointers are stored. The array is modified in runtime, e.g. for initialization. To make statical analysis more difficult, initialization of a function pointer variable and the call to it are placed in different blocks of code, i.e. different basic blocks. Additionally, they propose that dead code performing arbitrarily complex modifications to the array could be introduced to make the problem of statically determining the array's contents more difficult. The dead code would have to be protected by strong opaque predicates if it should be effective against static analysis.

We follow part of the proposal made by Wang et al. The addresses of each function used in direct calls are placed in an array local to the object file. Additionally this array is used for holding other pointer values, such as pointers to global values. Listing 3.2 shows an example of such an array.

Storing arbitrary pointer values in the array can hinder an attacker to disassemble the code into functions. Consider for example a local static function defined inside a file. When compiled into machine code this function will only consist of a sequence of instructions in a binary executable. In order to do the reverse transform, i.e. determining that the sequence of instructions form a function, a disassembler needs to find the start of the function.

There are two commonly used methods to perform disassembly: linear

```
1  int function_a();
2  int function_b();
3  int global_variable;
4
5  void (*function_array)(void)[5] = {
6    printf,
7    &global_variable,
8    function_a,
9    function_b,
10   function_a,
11 };
```

Listing 3.2. A global array for storing pointer values.

sweep and recursive traversal [25]. Adding bogus instructions before the start of the function can make disassembly with a linear sweep disassembler more difficult. The recursive traversal algorithm recursively disassembles reachable code, and if the function in question is reachable it may be correctly disassembled. This process is trivial for direct calls. For indirect calls however, the disassembler will have to perform static analysis to evalute the value of a function pointer in runtime, or fall back to guesses, in order to recurse to a function which only is accessed through indirect calls. If the array storing function pointers also contains pointers to non-functions, then the disassembler may make incorrect guesses about what is a function and what is not. For the sake of obfuscation, incorrect disassembly is a good thing as it makes the process of reverse engineering more troublesome for an adversary.

Each direct call to any of the functions stored in the array of function pointers is then replaced with an equivalent indirect call to that particular function through the function pointer stored in the global array.

As proposed by Wang et al., dead code is added for further obfuscation. The dead code is protected by opaque predicates and would change what function will be called if it was to be executed. An example of an obfuscated program is shown in Listing 3.3.

3.2.2 Function transformations

We define function transformations to mean inlining called functions (i.e. callees) into their caller, splitting a function in multiple parts, cloning a function into multiple copies, and combinations of these. Defining function transformations in this way means that if a platform has support for functions, it will inherently have support for function transformations. Collberg et al.

```
1  int function_a();
2
3  void (*function_array)(void)[] = {
4    ..., function_a, ...,
5  };
6
7  int main() {
8    int index = some_value;
9    ...
10   if (/* false opaque predicate */) {
11     index += /* arbitrary offset */
12   }
13   ...
14   /* call function_a */
15   int value = (int (*)())function_array[index]();
16   ...
17 }
```

Listing 3.3. An obfuscated call at line 16.

[4] first mentioned this type of obfuscation as *Inline and outline methods* and *Clone methods*. The motivation behind function transformations are described below.

The call graph of part of a program can serve as a tool for an adversary to understand the structure of a program. One target for the obfuscator is thus to hide information about the call graph. In the case of using indirect calls instead of direct ones, as outlined in Section 3.2.1, the result is an obfuscation of the edges in the call graph. Depending on the viewpoint, function pointer exploitation removes edges from the call graph or pollutes the call graph with lots of invalid edges. The nodes in the call graph are however kept intact. With function transformations, new nodes are created (due to cloning), other nodes are decoupled (due to inlining), some edges are collapsed (due to inlining), while other edges are added (due to splitting and inlining). The effect of these transformations is that the original call graph is not preserved. Thus some information is lost in this process. In particular an adversary will find it more difficult to relate multiple functions based on which functions they call.

Many of these transformations are likely to be touched by the optimizer when it is run on the code in a later phase. One of the optimizers most common tasks is for example to detect code that can be inlined. In the case of LLVM, the optimizer will inline code (if the optimization flags allow it) when an analysis finds that the cost of inlining the function is smaller than the benefit it adds [19].

Benefits from inlining include better cache locality and less overhead. The optimization applies the other way around as well, although less common in practice. An optimizer may decide that a piece of code should be split and pushed out into a new function instead. It may also recognize that two functions have equal semantics and replace them with a single function.

All function transformations with the purpose of obfuscating the code must hence have the optimizer in mind so that the transformations are not undone by the optimizer later on in the toolchain. To decrease the possibility of this we perform function transformations in three steps, as described in Listing 3.4.

```
for Function f in Program :
  # identify blocks of code that can be put into a new function
  # and perform this transformation randomly.
  f = splitFunction(f);

  # clone the resulting function in a random number of clones
  S = cloneFunction(f);

  for Function g in S :
    # inline a few of the calls in g, randomly
    g = inlineCalls(g);

  # update the callers to f to use one of the transformed
  # functions
  for Call c in callsTo(f) :
    h = random.select(S);
    c.setCalledFunction(h);
```

Listing 3.4. Pseudo code for the function transformation algorithm.

The original function is first split up into subfunctions, and the resulting function is cloned. Finally, some randomness is added to the functions through inlining. As there probably will be multiple copies of the function, this reduces the risk that the first split step is reversed by the optimizer. The last step ensures two things. Inlining is likely to collapse two nodes into one in the call graph. It also ensures that the cloned functions are not equal to each other so that static analysis is less likely to detect them as equal. This chance can be further reduced by performing other transformations on the resulting code.

It is essential that the original semantics of the code is preserved when performing function transformations. In particular, local variables shared by multiple instances of the function (i.e. local static variables in C) must continue to be shared. An obfuscator can ensure this by substituting the local variable with a shared global variable instead.

3.2.3 Exception exploitations

Drape [9] suggested that exceptions could be used for code obfuscation when it is supported by the programming language at hand. He proposed a couple of transformations from conditional branches to exceptional code. In particular he suggested that an obfuscation pass could perform the transformation from the code in Listing 3.5 to the code in Listing 3.6.

```
1 A;
2 B;
```

Listing 3.5. Two sequential statements.

```
1  try {
2    if (/* false opaque predicate */) {
3      throw error;
4    } else {
5      A;
6    }
7  } catch (error) {
8    /* bogus code */
9  } finally {
10   B;
11 }
```

Listing 3.6. Code which is semantically equivalent to the code in Listing 3.5, but which uses exceptions for control flow.

This approach, and variations of it have the benefit that the actual control flow is very similar to the original control flow. More specifically, the control flow will not contain exception handling as the exceptional branches are protected by opaque predicates.

Exception handling is very much dependent on the programming language and toolchain used. For example, the C++ ABI (Application Binary Interface) [3] defines how exception handling should work in C++. In particular the exception handling process consists of three phases:

1. Set up an exception handling object, containing information about the exception such as an object thrown and its type.

2. Raise the exception. The implementation unwinds the stack and decides where execution should proceed, i.e. which catch block that should handle the exception.

3. Control is passed to a relevant catch block, which may or may not trigger a new exception.

As the destination catch block is runtime dependent, there is no way to tell which catch block will resume execution without some static analysis on the type of the thrown object and the available catch blocks. Thus if the target ABI is the C++ ABI for an obfuscated build, an obfuscator can exploit the inherently complex exception handling mechanism for obfuscation purposes. One example of this is to change the control flow such that the normal control flow path will trigger exceptions and rely on correct exception handling, as shown in Listing 3.7.

```
try {
  if (/* true opaque predicate */) {
    throw error;
  } else {
    /* bogus code */
  }
} catch (error) {
  A;
} finally {
  B;
}
```

Listing 3.7. Code which is semantically equivalent to the code in Listing 3.6, but in which the normal control flow path will trigger exceptions.

There is a negative side effect of letting the exceptional path be part of the actual control flow path though. In particular exceptions usually have a large impact on the speed of a program. The exact impact varies between ABI's and implementations.

Our implementation combines transformations similar to those shown in Listing 3.6 and 3.7.

3.2.4 Value encoding

Collberg et al. [4] mentioned value encoding as a method to protect a program's values and constants from an adversary. Consider for example the code in Listing 3.8.

In the context of the code in Listing 3.8 the value 4412 is special and may be a target for an adversary. Value encoding is a transformation that conceals the constant while keeping the semantics intact. Collberg did not specify exactly how values ought to be encoded. In fact, it could make sense for an obfuscator to have multiple strategies for how a value is to be encoded,

```
1 int check_input(int input)
2 {
3     return input == 4412;
4 }
```

Listing 3.8. Sample code showing a function that validates the input.

depending on context, usage, and other factors. Consider for example the code given in Listing 3.9.

```
1 int check_input(int input)
2 {
3     int constant = 4412 ^ 1234;
4     return input == constant ^ 1234;
5 }
```

Listing 3.9. An example of value encoding. This code is semantically equivalent to the code in Listing 3.8

The code in Listing 3.8 and 3.9 are semantically equivalent. In this particular example the value encoding function is $\oplus$ with a constant value 1234. $\oplus$ is special in that it is the inverse of itself, but any injective function which works on the domain in question can be used. For the purpose of this report we identified two operators that can be used for encoding integer values.

- $\oplus$ – The second operand in the exclusive or operation can be randomly selected from the range of the first operand's type.
- $+$ – The second operand can be randomly selected from the range of the first operand's type. To do the inverse operation, arithmetic minus is applied. Note that this operator assumes that the definition of integer overflow has certain properties. This may or may not work depending on target platform and toolchain.

Naturally, integer types (including pointers) can be encoded with the method defined above. We do not consider floating point types for obfuscation in this report, as they have much more complex semantics. Below we specify how we defined the method to encode arrays and compound types. We also discuss how the obfuscator handles pointers to encoded variables.

Arrays

For simplicity, all elements in an array are encoded similarly. This means that the same function f is used to encode elements $A'[i] = f(A[i]), \forall i$.

Compound types

Each element in compound types, such as C structs and C++ classes, are encoded separately. Listing 3.10 shows an example of this.

```
struct Person {
    int age;
    int grade;
};

void foo() {
    struct Person bar;
    bar.age = 12 ^ 16237;
    bar.grade = 5 + 1231;
    printf("age: %d\ngrade: %d\n",
        bar.age ^ 16237, bar.grade - 1231);
}
```

Listing 3.10. An example of how compound types are obfuscated with value encoding.

Pointers to encoded variables

Many programming languages include support for pointers to arbitrary variables. It is possible that an encoded value is accessed through the means of a pointer, thus an obfuscator needs to be aware of possible pointer dereferences. If the obfuscator can determine what variable a pointer points to, value encoding can be applied to the value dereferenced, as shown in Listing 3.11.

```
int check_input(int input)
{
    int constant = 4412 ^ 1234;
    int *pointer = &constant;

    return input == *pointer ^ 1234;
}
```

Listing 3.11. An example of how value encoding can be applied when using pointers.

Another problem is the possibility that a pointer may point to two or more different variables depending on runtime conditions. There are at least two solutions to this problem.

- Check which variable the pointer points to in runtime and use a value encoding scheme appropriate for that variable.
- Apply the same value encoding scheme for all variables that may be dereferenced by the pointer in question.

As the first option does not scale very well if there are many different variables that a pointer can access, and because that option also would require extra code to determine what variable a pointer in fact points to, we chose the second option.

In order to determine the set of variables that may be dereferenced by a particular pointer we use a form of points-to analysis [1]. This analysis is also used for finding corner cases which are not supported by the obfuscator. Examples of such corner cases may include arbitrary pointer arithmetic, arbitrary casting between different types (for example accessing a 32-bit integer through a byte pointer) and passing pointers over function calls. An overview of part of the points-to analysis algorithm that we use for this is outlaid in Listing 3.12.

Algorithm overview

The value encoding algorithm uses the points-to analysis as described in 3.2.4 to determine two things: if encoding a variable is supported and what set of variables need to be encoded equally in order to keep the original semantics.

In case value encoding is supported on the variable in question, the algorithm adds code to decode the value resulting from each *load*, i.e. use of a value from an encoded variable. Likewise, code is added to encode a value before each *store*, i.e. assign of a value to an encoded variable. Constant values and initialization values are statically encoded. The idea is that when a value is stored in a variable, i.e. not during intermediary operations, the value should be encoded. When a value is loaded from a variable it is directly decoded.

3.3 Evaluation

For the evaluation we will use some of the metrics described in Section 2.6. More specifically we will use Halstead's metrics, cyclomatic complexity and nesting complexity to evaluate the effeciency of the obfuscations in terms of

```
/**
 * Points-to analysis of a value.
 *
 * @param value A value, i.e. a variable, that is to be analyzed.
 * @param level The pointer level. 0 means that a value has been
 * totally dereferenced. Dereferencing a variable decreases the
 * level, while taking the address of a variable increases the
 * level.
 */
int pointsToAnalysis(Value value, int level)
{
  for (Use use : uses(value))
  {
    if (!isSupported(use))
      return 0;

    if (isBinaryOperation(use)) {
      if (!pointsToAnalysis(use.getFirstOperand(), level) ||
        !pointsToAnalysis(use.getSecondOperand(), level))
        return 0;
    }
    else if (isTakeAddress(use)) {
      if (!pointsToAnalysis(use.getOperand(), level+1))
        return 0;
    }
    else if (isDereference(use)) {
      if (level == 1)
        // add this use to list of values that need to be decoded
      else if (!pointsToAnalysis(use.getOperand(), level-1))
        return 0;
    }
    else if (isAssign(use)) {
      if (level == 1)
        // add this use to list of values that need to be encoded
      else if (!pointsToAnalysis(use.getOperand(), level))
        return 0;
    }
    else ...
  }

  return 1;
}
```

Listing 3.12. Points-to analysis algorithm.

potency. Execution cost will be measured experimentally. We will reason about stealth, resilience and ease of integration.

We will perform the potency analysis as an intermediate step in the build process for a number of well known real world programs. The real world programs were chosen as follows: `tar`, `gzip` and `x264`. They were chosen because they were easy to access and benchmark. The analysis will first be done without obfuscations applied, then separately for each obfuscation method and finally with all obfuscation methods applied. The execution cost will be measured through running a number of test runs on the final executables. The description of the test which will be performed on each binary is described below:

tar Creation of an archive of reasonable size.

gzip Compression of a file of reasonable size.

x264 Encoding a 352x288 video clip with 300 frames to H.264 on default settings.

Additionally we will open the obfuscated files in *IDA* [24], which is a state of the art disassembler and debugger, and compare the results of the disassembly with the results when running IDA on the unobfuscated binary.

The binaries will be built with optimization enabled (equivalent to passing `-O2` to `clang`/`gcc`. These programs are crafted to take advantage of compiler optimizations. We want to compare the obfuscated versions of the program with its unobfuscated version built as it was intended to be built. Enabling optimizations for the obfuscated binaries will also give us insight into whether the compiler was able to remove the obfuscations through optimization. Moreover we may study the disassembled optimized obfuscated code to determine how the optimizer affected it.

Chapter 4

Implementation

In this chapter we discuss how the obfuscation methods were implemented in more detail. The chapter begins with a section about the tools used and why they were chosen.

4.1 Tools

Multiple tools were considered before the implementation proceeded. As the obfuscation methods considered for this report all consist of transformations that can be done on the source level of a program, tools for source-to-source transformations were considered first. In particular we looked at DMS Software Reengineering Toolkit [8], TXL [30] and Clang [18].

DMS's solution is proprietary software, which we wanted to avoid. The C++ support in TXL was deemed to be insufficient. Clang is both open source software and is considered to have very good support for C++. There is however no officially documented support for doing source-to-source transformations in Clang. As no source-to-source transformation utility that we looked at fulfilled our requirements, we looked at LLVM (see Section 2.3.3). In favor of LLVM there is a strong community, it is open source software, and there is documentation for writing transformation passes that work on LLVM IR. Additionally LLVM includes a set of analysis passes which we thought could be useful in the analysis of the obfuscation methods. Thus the implementation was written as an LLVM pass.

For analysis we also use a trial version of the IDA suite which is a disassembler and decompiler. It also provides a graphical interface for looking at the control flow graph and call graph of the disassembled program.

4.2 Obfuscator

As described in Section 4.1, the obfuscator was implemented as an LLVM pass. An LLVM pass is a process that works on the LLVM intermediary code for a so called module. A module may be the result of compilation of one or more input files. Each module contains information about what functions it contains, what global variables it defines and if applicable, what externally defined global variables and functions it depends on. It is also possible to write LLVM passes that operates on a set of functions, a single function and basic blocks. The LLVM pass manager handles the invocation and what dependencies between different passes there may be. The obfuscator was implemented as a module pass, because it may add, remove and modify arbitrarily many functions and global variables, something which is not supported for lower level passes.

4.2.1 Obfuscation methods

Similar to an LLVM pass, we define a common interface for obfuscation methods, shown in Listing 4.1. A user can specify what obfuscation methods should be available, and the obfuscator schedules these passes to run at the functions and global variables marked for obfuscation.

In order for a user to specify how a function or variable should be obfuscated, we define a set of options that could be passed to the obfuscator through annotations in the source code. In the current state, these options are simply an ordered list of obfuscation methods that should be applied by the obfuscator for the function or global variable at hand, but it can be extended if there is need for it.

4.2.2 Constant pool

Many obfuscation methods have a need for allocating global constants. Thus we create a pool for such constants, which all obfuscation methods have access to. The purpose of this pool is also to easily be able to obfuscate these constants without the need to write boilerplate code whereever such obfuscation might be needed.

4.2.3 Opaque predicates

Many obfuscation methods have a need for generating good opaque predicates. The strength of opaque predicates are considered dependent on what context is needed to calculate them. A constant opaque predicate is trivial

```
1  struct ObfuscationPass {
2    /**
3     * Initialize the pass to work on the given module.
4     *
5     * @param module The module that will be processed.
6     * @return true if initialization was successful.
7     */
8    bool initialize(llvm::Module &module);
9
10   /**
11    * Process the global given. This method is allowed to make
12    * arbitrary changes to functions and global variables.
13    *
14    * @param var The global variable to obfuscate.
15    */
16   void processGlobal(llvm::GlobalVariable &var);
17
18   /**
19    * Process the function given. This method is allowed to make
20    * arbitrary changes to functions and global variables.
21    *
22    * @param var The function to obfuscate.
23    */
24   void processFunction(llvm::Function &function);
25 };
```

Listing 4.1. An overview of the common interface for obfuscation methods.

to compute. An opaque predicate that can be calculated statically from local information in a function is considered weak. If non-local information is needed for calculating the opaque predicate however, the opaque predicate is considered to be better. Hence we think it is a good idea to integrate the opaque predicate implementation more tightly with the obfuscator.

4.2.4 Toolchain integration

To ease the integration of the obfuscator into an existing toolchain, the obfuscator is bundled with a script that can be used as a drop-in replacement for Clang. The drop-in replacement performs the required task in multiple steps. The enumeration below shows how it works in the normal case when a request for compilation to object code is requested by the user.

1. Compile the input to LLVM code with Clang.

2. Obfuscate the generated LLVM code with the obfuscator.
3. Generate assembly code for the target machine with the LLVM static compiler.
4. Compile the assembly code to object code with Clang.

Furthermore, optimization is requested for each step.

4.3 Testing

We verify the functionality of the implementation in three ways. First we check that the code is indeed obfuscated. For this we look at the generated code manually and in IDA. For checking that the obfuscation cover all corner cases, small test samples similar to unit tests will be used. The output from obfuscated builds are compared with the output from non-obfuscated builds. A similar process is also applied with larger, real world, programs.

4.4 Analysis

The information needed for the analysis is gathered with LLVM passes as well. More specifically there is one pass for determining the Halstead metrics, another for determining the cyclomatic complexity of a function and a similar pass for the nesting complexity. The results of the passes are written to a file on disk and then postprocessed in order to retrieve information concerning the whole program.

Chapter 5

Results

In this chapter we decribe the results of the evaluation on the obfuscation methods that were studied.

5.1 Potency and cost

In this section we describe the measured potency of the obfuscation methods. The results of the analysis are described in three different tables, one for each of the test programs; `tar` (Table 5.1), `gzip` (Table 5.2) and `x264` (Table 5.3). We then discuss the results. The results are presented as a ratio between the analysis of the obfuscated and the original version of the program. Below follows a description of the titles of the columns in the tables.

#Inst. The number of instructions in the generated LLVM IR code, i.e. a number which should be proportional to the number of lines needed to write it.

#Ops. The number of operands in the generated LLVM IR code, i.e. the number of values such as constants and variables in the code. Note that this refers to values in SSA code, thus one variable may generate multiple operands.

D. The Halstead difficulty metric, as described in Section 2.6.1.

Cycl. The cyclomatic complexity.

Nest. The nesting complexity.

Size The size of the generated main binary.

Mem. The peak memory usage during the computation of a common task.

Time The time measured to perform a common task.

It should be noted that the test only covers a subset of the programs. Most of the transformations applied also have some random element, such as what values are subject for encoding etc. Thus the numbers in the results should not be interpreted as exact, but rather as approximations.

Method	#Inst.	#Ops.	D.	Cycl.	Nest.	Size	Mem.	Time
Value enc.	2.40	2.10	0.84	1.00	1.00	3.82	1.10	1.36
Func. ptrs.	2.90	2.85	1.03	1.66	1.66	3.07	1.06	1.12
Func. transf.	1.88	1.88	1.05	1.98	1.96	1.63	1.02	1.01
Exc. expl.	1.28	1.29	1.04	1.28	1.28	1.35	1.05	170.87
All	23.62	23.68	1.51	5.20	5.10	40.09	1.59	187.03

Table 5.1. The results of the analysis on tar.

Method	#Inst.	#Ops.	D.	Cycl.	Nest.	Size	Mem.	Time
Value enc.	3.13	2.75	0.87	1.00	1.00	5.60	1.07	1.81
Func. ptrs.	1.84	1.83	0.99	1.37	1.37	1.88	1.02	1.00
Func. transf.	1.34	1.36	1.03	1.40	1.39	1.22	1.00	1.03
Exc. expl.	1.32	1.35	1.10	1.37	1.38	1.44	1.09	37.79
All	12.77	12.71	1.29	3.16	3.12	21.36	1.31	41.59

Table 5.2. The results of the analysis on gzip.

Method	#Inst.	#Ops.	D.	Cycl.	Nest.	Size	Mem.	Time
Value enc.	2.01	1.66	0.83	1.00	1.00	3.40	1.02	2.21
Func. ptrs.	2.72	2.50	1.27	1.32	1.32	3.93	1.01	1.01
Func. transf.	2.12	2.12	1.05	1.83	1.82	1.80	1.01	1.01
Exc. expl.	1.26	1.22	1.04	1.37	1.37	1.40	1.01	70.53
All	48.71	43.73	4.23	4.12	4.08	96.89	1.21	78.04

Table 5.3. The results of the analysis on x264.

We will look at each obfuscation method separately.

5.1.1 Value encoding

The value encoding obfuscation method increases the number of instructions by a multiple larger than 2 in all tests we performed. We believe that there are

multiple reasons for this. Value encoding adds a large overhead to each function as each value that is obfuscated generates at least one more instruction for loading the key needed to deobfuscate a variable and one instruction to perform the deobfuscation. The same applies when a value needs to be temporarily saved. The value encoding transformation also transforms constants to variables, which will cause a direct increase in the number of instructions and operands, but also an indirect increase due to less efficient optimizations such as constant propagation. Frequent obfuscation and deobfuscation in the code also cause other problems for the optimizer, such as when doing register allocation.

Halstead's difficulty metric shows a slight decrease in the complexity of the program. We believe that this metric is not suited for this kind of transformation as it merely looks at the number of instructions and operands. The value encoding obfuscation method adds many operands which are only used once, thus decreasing the ratio between total number of operands and the unique number of operands.

Regarding cyclomatic and nesting complexity, the result is expected. As value encoding does not change the control flow between basic blocks, these complexity measures should not be affected.

The slight increase in memory usage is probably due to increased need to spill registers to the stack as register allocation is more difficult with frequent encoding and decoding operations taking place. There are multiple reasons for the increase in time, e.g. less optimal optimizations, more overhead in terms of loading value encoding keys and performing decoding and encoding, etc.

Value encoding does not in itself affect the effectiveness of the IDA disassembler on the code. In combination with other obfuscation methods however, it may be able to hide information from the static analyzer regarding call targets for example.

5.1.2 Function pointer transformations

The function pointer transformation method does two things. Firstly it replaces direct calls with indirect calls. Secondly it adds dead code which modifies the value of the pointer used for the call. The dead code is protected by opaque predicates, which are added to the program. The amount of dead code added is proportional to the number of direct calls replaced.

From the results in Table 5.1, 5.2 and 5.3 suggests that the amount of code added by this transformation is quite big. The actual transformation from a direct call to an indirect call does not add many instructions or operands, but the introduction of dead code does so. We believe that the addition of blocks of dead code is the main source of the increase in all metrics except

memory and time usage. The slight increase in memory usage is probably due to reduced possibility to inline functions, thus forcing variables to be spilled on the stack. Except for `tar`, the performance drop is negligible, which was expected. Generally, the metrics measured for `tar` are high, and possibly the increase in time is due to reduced cache performance and overhead caused by the opaque predicates protecting the dead code.

The call graph of the obfuscated program shows that the transformation works. Almost all edges are absent, making it more difficult for an adversary to navigate in the program.

We had hoped that function pointer transformations would make it more difficult for the IDA disassembler to partition the code into functions. However, as we do not encode the pointers to the function entries, IDA is able to find references to the entries of the functions and thus marks the code as a function. As an experiment, we added encoding to the function pointers, and found that this made it more difficult for IDA to find references to the code. This would be a good feature, so we should consider working on encoding of the function pointers as part of future work.

5.1.3 Function transformations

We expected the outcome from applying function transformations to the code to be a modest increase in code size and performance decreases due to worse cache performance and unnecessary inlining. We see that the number of instructions increased approximately as much as the number of values, which is a consequence of that we are not adding new code, but merely cloning and inlining existing code. The cyclomatic and nesting complexities are increased slightly, due to the increased code size. We also see a slight increase in memory and time usage.

IDA is able to disassemble the code, but much information about which functions are related through a similar call is lost due to function cloning. Also, the call graph of the obfuscated program is much more complex than that of the original program. The increase in complexity is not caused by an increased number of nodes in the graph, but rather by the increased number of edges which probably is due to inlining.

5.1.4 Exception exploitation

As we have discussed, exceptional control flow paths are generally very expensive to follow. The increase in instruction count, operand count, difficulty, cyclomatic complexity, nesting complexity, binary size and memory usage are all quite modest. The slight increase in memory might be caused by the need

to allocate exceptional objects. As for the time usage, the results are disappointing but expected. We see performance drops in the range of 40–200 times. The transformation is randomized, and if an exceptional branch is added at a deep level in a loop for example, the performance will take a huge hit.

Interestingly, IDA is not able to bind together two pieces of code which are connected through an exceptional control flow path. An adversary would have to resort to other tools for static analysis in order to reconstruct the full control flow path.

Looking at the call graph, we confirm that the transformation works as expected. A few nodes concerning exception handling have been added to the graph, along with edges from the functions where normal branches have been replaced with exceptional branches.

5.1.5 All obfuscations

The order in which the obfuscation methods are applied in these tests is as follows:

1. Exception exploitation.
2. Function tranformations.
3. Function pointers.
4. Value encoding.

We wanted value encoding to be applied last because then the values introduced by other methods could be encoded as well. We also thought that it would be better to apply function pointer transformations after function transformations, as it would be difficult to do function transformations otherwise. It could be argued that exception exploitations should be applied after function transformations, as the control flow path between different versions of a cloned function then could be obfuscated differently. This does not matter however for the automated tests we perform.

As can be seen from looking at the results of the analysis, the obfuscation methods seem to strenghten each other, rather than just adding up. This effect is most apparent on Halstead's metrics, including the number of instructions and the size of the binary. Regarding the time usage, it seems like the exception exploitation obfuscation method still is the main cause of the slowdown. Thus it could be interesting to look at the results with all obfuscation

methods except exception exploitations applied. Due to time constraints we did not pursue this however.

The call graph in IDA is flattened due to the function pointer transformation. About 50% of the instructions in the fully obfuscated binaries are not assigned to a function, indicating that IDA is unable to disassemble the code or that it is unable to find suitable function entry and exit points. This should be compared to the number for the original binaries in which about 5–10% could not be classified correctly. This is an interesting observation, which shows that it is possible to hide code in a binary without much effort.

5.2 Resilience

As described in Section 2.6.2, resilience measures how easily an obfuscation can be removed. It is difficult to draw any final conclusions about the resilience, but we will discuss properties of the obfuscation methods that change how easily deobfuscation can be applied. We will discuss each of the obfuscation methods separately in the sections below. All reasoning below, if not otherwise stated, assumes that an adversary has been able to disassemble the code into some readable state.

5.2.1 Value encoding

Without any other obfuscation methods applied, value encoding is quite easily deobfuscated. In the most basic case, an adversary only needs to identify an obfuscated variable and the instructions for encoding and decoding the variable, and then simply remove those instructions. What may make the process of deobfuscation more difficult is constants which are statically encoded. A deobfuscator would then need to identify those constants and deobfuscate them statically. This is not a very difficult task in the simplest cases. However, if we add other obfuscation methods such as opaque predicates and dead code to the picture, deobfuscating value encoding transformations will become much more difficult.

Consider for example the piece of code in Listing 5.1. If we assume that the adversary is not able to identify the opaque predicates here, it will be impossible to draw any conclusions regarding how the variable *var* is encoded.

5.2.2 Function pointer transformations

A constant static function pointer used in an indirect call is trivial to deobfuscate. However, if this pointer is obfuscated in such a way that it is not trivial

```
1  int var;
2
3  if (/* true opaque predicate */)
4      var = 50 ^ 567;
5  else
6      var = 60 ^ 787;
7
8  if (/* false opaque predicate */)
9      printf("%d\n", var ^ 787);
10 else
11     printf("%d\n", var ^ 567);
```

Listing 5.1. An example of the difficulty of deobfuscation when value encoding is combined with opaque predicates.

to determine its value statically, then the resilience will be higher. There are multiple ways to do this. We use opaque predicates to decide what function pointer to use, and we also mix function pointers and ordinary pointers so that an adversary will not only have to determine what control flow path will be used, but also if a particular pointer is a pointer to a function or if it is a pointer to something else. On top of this we can add value encoding obfuscations. With the two methods combined, an adversary needs to reverse both transformations before the code has been fully restored. As we discussed in Section 2.5.3, the problem of statically determining which function is called in general is NP-hard. We believe that this increases the resilience value of this obfuscation method, although our particular implementation might not have this property.

5.2.3 Function transformations

Cloning a function is a reversible process. Inlining of a function is however much more difficult to revert, especially when the code has been optimized afterwards. Even though an adversary knows that two sets of basic blocks originally corresponded to a single function, the process of replacing these two sets of blocks with a single function might be very difficult due to assumptions about the context which could be made when the code was inlined and optimized.

5.2.4 Exception exploitation

As noted in Section 3.2.3 the transformation from an ordinary branch to an exceptional branch is basically one-to-one. This makes it very easy to write an obfuscating transformation, but it also makes it quite easy to write a transformation that deobfuscates the code. We note that two things can make this more difficult: opaque predicates and the exception handling ABI.

We protect exceptional branches with opaque predicates. As noted earlier, it may be difficult to draw conclusions about opaque predicates statically. Also, opaque predicates merely hide the real control flow path, they do not hide the fact that an exception might occur.

Regarding the exception handling ABI of C++, there is no trivial direct connection between the try block and the catch block. When an exception is thrown, the stack is unwound and external code is used to find the code to jump to. An adversary would have to simulate this external code and draw conclusions about what would happen in case the exception was thrown (i.e. if there are multiple catch blocks) and then replace the exceptional code path with a simple branch. Depending on what the exceptional control flow path generated by the obfuscator looks like, the difficulty of this process will vary.

5.3 Ease of integration

In this section we argue about the difficulty to implement and integrate each of the studied obfuscation methods into an existing toolchain.

5.3.1 Value encoding

The difficulty to implement value encoding is very much dependent on how advanced the methods employed are. There are many corner cases to consider while implementing value encoding, e.g. encoding across function calls and whether the code is in a threaded environment. Value encoding is also dependent on correct and reasonably efficient points-to analysis, as we described in Section 3.2.4.

This obfuscation method does not require any extra support in the programming language or target platform, thus we deem it easy to integrate in an existing toolchain.

5.3.2 Function pointer transformations

Function pointer transformations depends on the support for indirect function calls in the programming language. It needs to be complemented with other

obfuscation methods to be effective, methods which may be difficult to implement. The implementation of the actual transform from direct to indirect calls is however from our experience easy.

5.3.3 Function transformations

Practically all programming languages have support for functions, and function transformations such as cloning, inlining and splitting does not depend on any other programming language feature. The actual implementation of cloning, inlining and splitting is not trivial however, and may be very difficult in specific circumstances. We used existing compiler infrastructure to perform these tasks, but even so it was non-trivial to yield a correct result under all circumstances.

5.3.4 Exception exploitations

Of all obfuscation methods we studied, exception exploitations is probably the method with most dependencies on the programming language and target platform. Not all programming languages have support for exceptions, e.g. C. Further more, depending on what level the obfuscation method is implemented (source level, intermediate representation level, etc), the required amount of compiler and platform specific information varies. As we have described in Section 5.1.4, exception exploitations come with a big performance hit. Thus this method may be inapproppriate in parts of a system where performance matters.

5.4 Stealth

In this section we briefly argue about the stealth perspective from our findings during this study. The arguments heavily relies on hands on experience with the code produced with our implementation, and some of the issues may be possible to hide through clever tricks. This is left as future work.

5.4.1 Value encoding

In order to make it more difficult for the compiler, as well as an adversary, to do constant propagation optimization on the value encoding code, the value encoding transformation produces code which heavily relies on global variables. Thus, code which has been transformed with this obfuscation method will make use of many global variables along with simple revertible operations

such as exclusive or (**XOR**). Given the knowledge of this fact, it is quite easy for an adversary to spot that a piece of code has been obfuscated with value encoding. From our knowledge, it is not that common to have functions which load 10 or more different global variables which sole purpose is to be the second operand in simple arithmetic operations. We deem the stealth property of value encoding low compared to the other obfuscation methods that we studied. Higher stealth can be accomplished through more clever ways to encode and decode values, or through simply decreasing the number of values subject to encoding.

5.4.2 Function pointer transformations

The commodity of function pointers is heavily dependent on the context, i.e. the program being obfuscated. Assuming that an adversary has little information about the original source code, the sole existence of function pointers is not information enough to conclude that the program or part of it has been obfuscated. If almost all function calls in some context such as a function however are obfuscated, then such conclusions might be more probable. As we argued in Section 5.2.2, function pointer transformations should be considered to be combined with other obfuscation techniques, such as value encoding and dead code. If that is the case, then stealth will be dependent on those methods as well. However, function pointer transformation in itself and applied with care, we deem reasonably stealthy.

5.4.3 Function transformations

Function transformations simply change the number of functions present in the program, and the size of those. They do not add any new code. Also, function transformations could be written in such a way that they do not alter the total number of functions. Furthermore, the number of functions and the size of them varies much from program to program, so it would be difficult to find the existence of this transformation from those metrics. One of the main purposes of function transformations is that to modify the call graph tree. An adversary might look at the metrics concerning number of calls per function and so forth, but again – these numbers probably vary much depending on the context. Also, combined with other obfuscation methods such as function pointer transformations, the task of mesuring the number of calls per function statically could prove difficult. We think that the stealth for the function transformations obfuscation method is high compared to the other methods we studied.

5.4.4 Exception exploitation

Normally, exceptions are used for what they are, i.e. to add a control flow that should only happen in exceptional cases. In the case of exceptional exploitations obfuscation, exceptions are used in the normal control flow. An adversary could intercept which branches are taken dynamically and conclude that some of these exceptional paths are just obfuscation. An exceptional branch is also easy to spot through the use of calls to the standard library. Additional obfuscation would be needed to hide the fact that those methods are called. However, used sparsely, exceptional exploitations may be difficult to spot for an adversary. In summary we think that exception exploitations is an obfuscation method which if used wisely has mid range stealth.

5.5 Summary

In this section we summarize the findings in our evaluation of the obfuscation methods implemented. We do this by giving an approximate rating of each of the obfuscation methods studied concerning potency, resilience, stealth and ease of integration. We base these ratings on the discussion outlayed earlier in this chapter. The ratings are shown in Table 5.4. Note that, as discussed, the obfuscation methods may be especially useful in combination with other methods. This property is marked with a + as suffix to the rating. The ratings below are given as integers in the range $[1, 5]$ were higher is better.

Method	**Potency**	**Cost**	**Resilience**	**Stealth**	**Ease**
Value encoding	3+	3	2+	1	2
Function pointers	4	5	4	5	5
Function transf.	3+	5	4+	4	3
Exception expl.	3	1	3	3	3

Table 5.4. A summary rating of each of the obfuscation methods studied in this paper. Ratings range from 1 to 5 where higher is better.

It would seem that function pointers and function transformations are the most efficient obfuscation methods in this study. We argue that they are, because they hide much of the original control flow information from an adversary. Value encoding is also useful, and more so for improving the potency and resilience of information leaked by other obfuscation methods. Exception exploitation is an interesting method, but alas it comes at a very high price. It can further be argued that as the transform is almost one-to-one, it can be undone easily with sufficient static or dynamic analysis.

Chapter 6

Discussion

In this chapter we discuss what we learned, what we can conclude from the results, what our more subjective conclusions are and also examples of topics for future work in this area.

6.1 Conclusions

In this report we have studied a number of code obfuscation techniques which can be implemented in a platform independent way. We have evaluated them in terms of potency, resilience, stealth and ease of integration. The implementation was based on the open source LLVM infrastructure. As the obfuscation methods were implemented as transforms on a quite high level language, we argue that it should be possible to port them to multiple platforms. For improved platform independence source to source transformations would serve. Another solution would be to write a C back end for LLVM.

Though code obfuscation may sound difficult at first, there are many tools available that make it easy to get started with simple transformations. Some of the obfuscation methods discussed in this paper are difficult to implement in a generic way, some of them are not. We note that some of the easier methods still yield very good results in terms of potency and resilience.

6.1.1 Theoretical Results

We evaluated four different code obfuscation techniques more thoroughly, namely value encoding, function pointer transformations, function transformations and exception exploitations. Our evaluation showed that function pointer transformations and function transformations are the most efficient obfuscation methods in terms of the average of potency, cost, resilience, stealth

and ease of integration, as shown in Table 5.4. Value encoding proved potent, but lacked in terms of stealth. Exception exploitations were shown to be very costly, although their efficiency in terms of other criteria were good.

The obfuscation method we consider the most difficult to implement was value encoding, as there are so many special cases to care about. Function transformations are easy conceptually, while much more difficult to implement in LLVM IR due to the added layers of details.

On the negative side, we think that the metrics we chose for evaluating the obfuscation methods did not measure the potency sufficiently well. In particular we think that Halstead's difficulty metric did not reflect the complexity of the program accurately. Some features of the obfuscation methods were not measurable at all, such as the added complexity of using indirect calls instead of direct ones.

Originally we intended to manually evaluate the obfuscation methods, but due to time constraints this was omitted. In retrospect we think this would have improved the credibility of the analysis in all areas.

6.1.2 Subjective Thoughts

Personally when I try to figure out how a certain program works, I start by structuring the part of the program I am interested in into smaller parts. Each part can then be assessed in detail if need be and the smaller parts can together give a better understanding of what a larger part does, and ultimately a deep knowledge about the whole program may be gathered. This assumes that I can understand what smaller parts make up the bigger part.

Function transformations, function pointer transformations and exception exploitations all make this process more difficult by both introducing new edges between parts of the program and hiding other edges. As we have covered elsewhere, this process may be reversed – but that process in itself may be time consuming and difficult.

What these methods do well is to make it more difficult for an attacker to understand what part of the program should be analyzed in the first place. All obfuscations can be removed with enough determination, but if the program is large enough and well obfuscated that task may be too time consuming for anyone to succeed.

Value encoding provides mostly local obfuscation. By itself I do not think it is a very good obfuscation method. As a layer on top of other obfuscation methods however I think it can make code much more difficult to understand and deobfuscate.

An issue with obfuscation in general is that if only part of the program is obfuscated, it may give away information about what part of the program that

is considered most valuable. This is mostly a problem if only value encoding is used as it does not change the structure of the program. This assumes that the adversary can find what part of the program is obfuscated which is dependent on the stealth of the implementation used. Some compilers are known to produce quite obfuscated binary code in the first place, so this task may be more difficult than it sounds.

The code produced by the obfuscator written as part of this work is indeed more difficult to understand than the original code. The code obfuscated with value encoding is easy to spot. The frequent use of function pointer transformations provides less stealth, but it is time consuming and error prone to dereference the function pointers. The function transformations, especially the inclusion of dead code, is easy to spot and could be improved. As we have stated in the report, good opaque predicates is necessary for this type of obfuscation to work well and this is an area in which the obfuscator could be improved. The obfuscation that is most potent is the exception exploitation obfuscation method. Although we could argue about the stealth of this method in our implementation, we find the code obfuscated in this way very difficult to read.

6.2 Future work

It would be interesting to further improve the work on value encoding in particular with regards to threading and encoding variables over function calls.

We leave it as future work to review other analysis methods which are able to give a more accurate measurement of the potency. In particular a method that takes into account the added obscurity of indirect function calls would be useful. Stealth and resilience are criteria that also need better and less subjective metrics which can be used for evaluation.

Another area that should be considered for future work is the ordering of obfuscation methods. This was something that Collberg pointed out in his first paper and to the best of our knowledge there has since been no progress in this area.

Bibliography

[1] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". DIKU Research Report 94/19. PhD thesis. DIKU, University of Copenhagen, Denmark, 1994, p. 113.

[2] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. "On the (Im)possibility of Obfuscating Programs". In: *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. 2001.

[3] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. *Itanium C++ ABI*. (accessed May 28th 2012). URL: `http://sourcery.mentor.com/public/cxx-abi/abi.html`.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1998, pp. 184–196.

[6] Keith Cooper and Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.

[7] Microsoft Corporation. *Microsoft Visual Studio*. (accessed May 28th 2012). URL: `http://www.microsoft.com/visualstudio/en-us`.

[8] Semantic Designs. *The DMS Software Reengineering Toolkit*. (accessed May 28th 2012). URL: `http://www.semdesigns.com/products/DMS/DMSToolkit.html`.

[9] Stephen Drape. *Intellectual Property Protection using Obfuscation*. Tech. rep. RR-10-02. Department of Computer Science, University of Oxford, 2010.

[10] Python Software Foundation. *The Python Language Reference*. (accessed April 16th 2012). URL: `http://docs.python.org/reference/index.html`.

[11] GNU. *GNU Compiler Collection*. (accessed May 28th 2012). URL: `http://gcc.gnu.org/`.

[12] Huisheng Gong and Monika Schmidt. "A complexity measure based on selection and nesting". In: *Performance Evaluation Review* 13.1 (1985), pp. 14–19.

[13] James Gosling and Henry McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, 1996.

[14] PARIS research group. *Diablo*. (accessed April 26th 2012). URL: `http://diablo.elis.ugent.be/`.

[15] Warren A. Harrison and Kenneth I. Magel. "A complexity measure based on nesting level". In: *SIGPLAN Notices* 16.3 (1981), pp. 63–74.

[16] Tommy Hoffner. *Evaluation and Comparison of Program Slicing Tools*. Tech. rep. Department of Computer and Information Science, Linköping University, Sweden, 1995.

[17] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988.

[18] Chris Lattner. *Clang*. (accessed May 28th 2012). URL: `http://clang.llvm.org/`.

[19] Chris Lattner. *Low Level Virtual Machine*. (accessed May 28th 2012). URL: `http://llvm.org/`.

[20] MTC Group LTD. *Morpher*. (accessed April 26th 2012). URL: `http://morpher.com/`.

[21] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. "Positive Results and Techniques for Obfuscation". In: *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques*. 2004, pp. 20–39.

[22] Thomas J. McCabe. "A complexity measure". In: *Proceedings of the 2nd international conference on Software engineering*. 1976, pp. 407–.

[23] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. "Software Obfuscation on a Theoretical Basis and Its Implementation". In: *IEICE Trans. Fundamentals* E86-A (2003-01-01), pp. 1–11.

[24] Hex-Rays SA. *IDA*. (accessed May 20th 2012). URL: `http://www.hex-rays.com/products/ida/index.shtml`.

[25] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. "Disassembly of Executable Code Revisited". In: *Proceedings of the Ninth Working Conference on Reverse Engineering*. 2002, pp. 45–.

[26] Monirul Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. "Impeding Malware Analysis Using Conditional Code Obfuscation". In: *Proceedings of the 15th Annual Network and Distributed System Security Symposium*. 2008.

[27] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Addison-Wesley Longman Publishing Co., Inc., 2000.

[28] Oreans Technology. *Themida*. (accessed April 26th 2012). URL: `http://www.oreans.com/themida.php`.

[29] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. "Deobfuscation: Reverse Engineering Obfuscated Code". In: *Proceedings of the 12th Working Conference on Reverse Engineering*. 2005, pp. 45–54.

[30] Queen's University. *TXL*. (accessed May 28th 2012). URL: `http://www.txl.ca/`.

[31] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Tech. rep. CS-2000-12. Department of Computer Science, University of Virginia, USA, 2000.