The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

LAMBDA OBFUSCATION

A Thesis in
Information Sciences and Technology
by
Pengwei Lan

© 2017 Pengwei Lan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2017

The thesis of Pengwei Lan was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
Thesis Advisor

Danfeng Zhang
Assistant Professor of Computer Science and Engineering
Committee Member

Zihan Zhou
Assistant Professor of Information Sciences and Technology
Committee Member

Andrea H. Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

With the rise of increasingly advanced reverse engineering technique, especially more scalable symbolic execution tools, software obfuscation faces great challenges. Branch conditions contain important control flow logic of a program. Adversaries can use powerful program analysis tools to collect sensitive program properties and recover a program's internal logic, stealing intellectual properties from the original owner. In this thesis, we propose a novel control obfuscation technique that uses lambda calculus to hide the original computation semantics and makes the original program more obscure to understand and reverse engineer. Our obfuscator replaces the conditional instructions with lambda calculus expressions that simulate the same behavior with a more complicated execution model. Our experiment result shows that our obfuscation method can protect sensitive branch conditions from state-of-the-art symbolic execution techniques, with only modest overhead.

Table of Contents

List of Figures vi

List of Tables vii

Acknowledgments viii

Chapter 1
Introduction 1

Chapter 2
Related Work 4

Chapter 3
Design 7
- 3.1 Lambda Calculus Basics . . . 8
  - 3.1.1 Reduction . . . 9
  - 3.1.2 Church Encoding . . . 10
- 3.2 Data Structures . . . 13
- 3.3 Lambda Obfuscation . . . 15

Chapter 4
Implementation 17
- 4.1 Stage One: Preprocessing . . . 19
- 4.2 Stage Two: Transformation . . . 19
  - 4.2.1 Identifying Instruction Candidates . . . 20
  - 4.2.2 Transforming Instruction . . . 20
- 4.3 Stage Three: Compilation . . . 21

Chapter 5
Evaluation 22
- 5.1 Potency . . . 23

5.2 Resilience 24
5.3 Cost 25
5.4 Stealth 27

Chapter 6
Discussion **29**
6.1 Countering Dynamic Monitoring 29
6.2 Potential Extensions 30
6.3 Combining with Other Obfuscation 31
6.4 Obfuscating Complete Branch Predicates 31

Chapter 7
Conclusion **33**

Bibliography **34**

List of Figures

3.1 SUCC operator and Church numeral 2 in term structure 14

4.1 The workflow of lambda obfuscation 18
4.2 LLVM IR code of a C program before and after obfuscation 18

5.1 C program to be obfuscated in KLEE experiment 25
5.2 Instruction distribution of bzip2 27
5.3 Instruction distribution of regexp 28

List of Tables

3.1 C data structure: term 14

5.1 Program metrics before and after obfuscation at obfuscation level 30% 24
5.2 Overhead of lambda calculus obfuscation on bzip2 and regexp . . . 26

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Dinghao Wu for his continuous support of my Master study. Without his knowledge, guidance and encouragement, this thesis would not haven been possible for me. It has been a great honor to be his student and learn from him. I could not ask for a better advisor and mentor for my Master study.

I would like to thank Dr. Danfeng Zhang and Dr. Zihan Zhou for serving as my committee members. I am thankful to them for sharing their insightful comments and suggestions.

I would also like to acknowledge my fellow labmates, Pei Wang and Shuai Wang for their precious cooperation and support. I am indebted to them for their passionate participation and input that makes this thesis research successful.

Finally, I would like to thank my family. None of this could have happened without my family. Thank you.

This research is supported in part by the Office of Naval Research (ONR) under grant No. N00014-16-1-2912 and the National Science Foundation (NSF) under grant No. CCF-1320605.

Chapter 1

Introduction

With the notable advancement of binary analysis techniques, reverse engineering is becoming more effective than ever before. As a result, malicious parties are able to employ the latest binary analysis tools to identify software vulnerabilities and exploit them to inject malicious codes into legit applications. Binary analysis tools can also get misused to reveal important internal logic of the distributed software copies, potentially leading to intellectual property thefts and therefore severe financial loss to the original developers.

One of the protection techniques that prevents undesired reverse engineering is software obfuscation. Generally, software obfuscation is the program transformation that makes software more complicated than its original form and difficult for adversaries to understand and analyze, while preserving the program's original semantics [1].

In this thesis, we propose a novel obfuscation method, called lambda obfuscation, which utilizes the concept of lambda calculus, a powerful formal computation system widely adopted by the programming language community. The main idea of our approach is to utilize the unique computation model of lambda calculus, which is vastly different from the widely used imperative programming paradigm, to simulate the computation of certain parts in the original programs. Instead of performing computation with data and control in imperative programming, lambda calculus is entirely based on function application and reduction. The concept of control flow becomes insignificant in lambda calculus, and all data structures, including primitive data types like integers, can be represented as high-order functions, potentially making conventional information flows implicit. With this highly abstract computation model implemented by the low-level machine code, there will be a huge semantics gap that imposes great challenges on automated program analysis and reverse engineering. Being Turing complete and considered as the smallest universal programming language [2], lambda calculus is capable of using its reduction rules to express any kind of imperative computation. If the simulated computation is free of side effects, the source-level conversion can be quite straightforward, yet the resulting program binary after transformation will become much more complicated and obscure.

To demonstrate the value of our technique, we have implemented a prototype of the lambda obfuscation based on LLVM [3]. The obfuscator can transform qualified conditional instructions into corresponding function calls that simulate original behavior using lambda calculus. An interpreter that evaluates lambda calculus is linked to compiled binaries to return correct signals that can guide following conditional jump instructions. In such way, the behavior of conditional instructions

is preserved while the execution is through lambda calculus to increase the complexity. We evaluate our obfuscation technique in four aspects, namely potency, resilience, cost and stealth. The evaluation result shows that our method can make the obfuscated programs more obscure and prevent automatic software analyzers from revealing possible execution paths. In particular, we assessed lambda obfuscation's resilience against KLEE, an advanced symbolic execution engine [4] and obtained promising results.

The rest of the thesis is organized as follows. We first discuss related work on control obfuscation in Chapter 2. We then briefly introduce the basics of lambda calculus, followed by the design of lambda obfuscation in Chapter 3. The technical details of the implementation are presented in Chapter 4. Chapter 5 presents the evaluation result of our approach. Some research questions are discussed in Chapter 6 and we finally conclude the thesis in Chapter 7.

Chapter 2

Related Work

Software obfuscation techniques can be divided into four major categories: layout obfuscation, preventive obfuscation, data obfuscation, and control obfuscation [5]. Arguably as the most popular one, control obfuscation focuses on concealing and complicating control flow information of the program. There has been diverse research striving to develop effective control obfuscation techniques through different angles.

One of the classic approaches to develop control obfuscation technique is to design resilient opaque predicates. An opaque predicate is a predicate that always evaluates to True or False, which is only known to the creator of the program but difficult for adversaries to identify [6]. Most of these opaque predicate are derived from algebraic theorems [7] or quadratic residues [8]. However, one of the fundamental flaws of opaque predicates is that opaque predicates always evaluate to the same value during multiple runtime. The invariant nature of opaque predicates can

result in a likely detection by adversaries through sophisticated program analysis. In order to overcome this disadvantage of invariant opaque predicates, Palsberg et al. [9] introduced dynamic opaque predicates in which a family of correlated predicates retain to the same value in one given execution but change the evaluation value in another run.

Sharif et al. [10] proposed a conditional code obfuscation technique that leverages the inconvertibility of hash function to protect the conditional branch. In their study, they used the hash function to obfuscate the value of variable for which the branch condition can be satisfied. Because of the cryptographic characteristics of hash function, it is not feasible for code analyzers to reconstruct the values that satisfy the condition and the control flow logic information is therefore concealed and protected. However, their approach is only effective in protecting equality operator while not suitable for obfuscating relational operators that check ranges of values, such as larger than, smaller than or not equal to.

There are also multiple research studies building their obfuscation techniques based on code mobility [11, 12, 13]. These obfuscation approaches only deploy partial and incomplete application code on the local machine and retrieve the rest of binary instructions from a remote trusted server. While these obfuscation techniques can reduce attacker's visibility to the software semantics, they also severely rely on the availability of remote servers which decreases the feasibility of their techniques.

Control flow obfuscation can also be implemented using neural networks. Ma et al. [14] proposed to replace important branch conditions with trained neural

networks that simulate the program behavior when the branch conditions are triggered. Their approach can protect program against concolic testing due to the complexity of neural networks. However, it is required to train corresponding neural networks in advanced based on the target branch conditions. Their approach becomes less flexible and tedious to deploy when the number of branch conditions requiring obfuscation increases.

Wang et al. introduced another obfuscation framework called transligual obfuscation [15]. They proposed to embed a different programing language into the original program by translating partial of the source code. Because the original languages and embedded language have different programming paradigms and execution models, translingual obfuscation can make the program more obscure for attackers to understand and reverse engineer. Our research shares a similar idea with transligual obfuscation in that we both propose to leverage the execution model of a different programming language.

Chapter 3

Design

The basic idea of lambda obfuscation is to leverage the unique computation model of lambda calculus for protecting the relatively straightforward imperative computation procedures in common programs. While different programming languages have different execution models, imperative languages whose computation scheme aligns the best with the underlying hardware are considered to be relatively easier to reverse engineer. Typical imperative languages include C, Fortran, and Pascal. On the contrary, execution models of functional languages, such as lambda calculus, result in greater differences between the source code and compiled binary code, which can enhance the difficulty of de-obfuscation. Therefore, we can translate and implement functionalities of a program using different programming language to mix two execution models and conceal sensitive program information. In this thesis, our obfuscator embeds functional execution model of lambda calculus into C programs that use imperative execution model. It translates the path condition instructions in original compiled binary code into function calls that are

implemented using lambda calculus. In this way, we are able to make the execution model of the obfuscated programs more complicated, thus hindering reverse engineering.

3.1 Lambda Calculus Basics

Lambda calculus is a formal system that uses the basic operations of function abstraction and application to describe all computation [16]. The basic building blocks of lambda calculus are expressions called lambda terms. There are three types of lambda terms, namely *variable, abstraction and application*, the syntax of which is defined by the following BNF specifications:

$$
\begin{array}{rcll}
\langle \mathit{expression} \rangle & ::= & \langle \mathit{variable} \rangle & \textbf{Variable} \\
 & | & \lambda \langle \mathit{variable} \rangle . \langle \mathit{expression} \rangle & \textbf{Abstraction} \\
 & | & \langle \mathit{expression} \rangle \langle \mathit{expression} \rangle & \textbf{Application}
\end{array}
$$

A *variable* in lambda calculus is an arbitrary identifier. An *abstraction* can be viewed as a notation for defining anonymous functions. For example, lambda term $(\lambda x.e)$ defines an anonymous function whose parameter is *variable* x and function body is *expression* e, that can be another valid lambda term. An *application* captures the action of applying a function to its arguments. For example, lambda expression $(x\ y)$ represents applying function x to an input parameter y. All the other valid lambda terms can be formed by repeated combinations of the three

basic lambda terms. The following are some examples of valid lambda terms:

$$x$$

$$\lambda x.x$$

$$(\lambda x.x)\ y$$

$$\lambda p.\lambda q.p\ q\ p$$

When the λ symbol precedes a variable, it binds all the occurrences of this variable in the abstraction body. A variable is called *bound variable* if it is bound by a λ symbol. Other variables in the function body are called *free variables* [17]. For example, in the following expression, variable x is a bound variable while variable y is a free variable.

$$\lambda x.x\ y$$

3.1.1 Reduction

The meaning of lambda calculus is defined by how lambda calculus can be reduced [18]. This reduction process is achieved by substituting all free variables in a way similar to passing the defined parameters into the function body during a function call [19]. The main rule to perform reduction in lambda calculus is called β-reduction, which can be defined as follows:

$$(\lambda x.e_1)\, e_2 \Rightarrow e_1[x \rightarrow e_2]$$

where notation $e_1[x \to e_2]$ indicates substituting the argument e_2 for all free occurrences of the variable x in body e_1. β-reduction captures the essence of function application and can be used to simplify and evaluate lambda terms. During the reduction process, all intermediate function applications are carried out and eliminated. The reduction process stops when β-reduction rule cannot be performed any more. Here are several β-reduction examples.

$$(\lambda x.x)\ y \Rightarrow y$$

$$(\lambda x.x)(\lambda y.y) \Rightarrow \lambda y.y$$

$$(\lambda x.x\ x)(\lambda y.y) \Rightarrow (\lambda y.y)(\lambda y.y) \Rightarrow \lambda y.y$$

3.1.2 Church Encoding

In lambda calculus, all expressions can be considered as functions, which means function is the only primitive data type that lambda calculus originally supports. To perform arithmetic calculation using lambda calculus, it is imperative that we encode arithmetic operators and integeral numbers in the obfuscator. Lambda calculus is capable of representing all computable operators along with its operands [20]. In this thesis, we employ Church encoding to encode natural numbers and operators to implement lambda obfuscation.

The idea of Church numerals is to count how many times a function has been applied to a value. For example, Church numeral 2 means a function is applied to

a input value twice. In Church encoding, Church numerals are defined as follows:

$$0 \equiv \lambda f.\lambda x.x$$
$$1 \equiv \lambda f.\lambda x.f\ x$$
$$2 \equiv \lambda f.\lambda x.f\ (f\ x)$$
$$3 \equiv \lambda f.\lambda x.f\ (f\ (f\ x))$$
$$n \equiv \lambda f.\lambda x.f^n\ x$$

As the definition indicates, Church numeral n can be viewed as a high-order function that takes a input function f and applies it to a value x for n times. Therefore, a successor (SUCC) operator that takes a Church numeral n and returns n + 1 essentially is appending another application of function f to Church numeral n, which is defined as follows:

$$\text{SUCC} = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

While adding m to n is equivalent to adding 1 to n for m times, a PLUS operator that adds m to n is identical to applying SUCC operator to n for m times. Therefore, PLUS operator can be defined using SUCC operator as follows:

$$\text{PLUS} = \lambda m.\lambda n.m\ \text{SUCC}\ n$$

The predecessor (PRED) operator that takes a Church numeral n and returns $n - 1$ is more complicated to define, but conceptually it is still equivalent to getting the high-order function that applies its argument one less time than Church

numeral n. Similarly, subtraction (SUB) operator can be defined based on PRED operator. Other important operators and logical predicates are defined as follows:

$$
\begin{aligned}
\text{PRED} &= \lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u) \\
\text{SUB} &= \lambda m.\lambda n.n\ \text{PRED}\ m \\
\text{TRUE} &= \lambda x.\lambda y.x \\
\text{FALSE} &= \lambda x.\lambda y.y \\
\text{ISZERO} &= \lambda n.n\ (\lambda x.\text{FALSE})\ \text{TRUE} \\
\text{LEQ} &= \lambda m.\lambda n.\text{ISZERO}(\text{SUB}\ m\ n) \\
\text{GEQ} &= \lambda m.\lambda n.\text{LEQ}\ n\ m
\end{aligned}
$$

Through implementing the Church encoding of necessary operators, we are able to perform basic arithmetic operations in lambda calculus, like plus, subtraction, multiplication and divide and compare operations such as larger than, equal to. For example, $0+1$ is equivalent to perform reduction on the following lambda term in lambda calculus:

$$
\begin{aligned}
\text{PLUS}\ 0\ 1 &\equiv (\lambda m.\lambda n.m\ \text{SUCC}\ n)(\lambda f.\lambda x.x)(\lambda f.\lambda x.f\ x) \\
&\equiv (\lambda m.\lambda n.m\ (\lambda n.\lambda f.\lambda x.f\ (n\ f\ x))\ n)(\lambda f.\lambda x.x)(\lambda f.\lambda x.f\ x)
\end{aligned}
$$

In other words, the Church encoding provides the lambda calculus functions we can replace path condition instructions with. Our obfuscator also implements Church numerals so both operands of instructions can be encoded into Church numerals to support the computation.

From the obfuscation perspective, the Church encoding “accidentally” possesses the capability of eliminating explicit control flows. As an example, the ISZERO lambda term simualtes a typical branch operation in imperative programming. However, the computation, or more precisely the reductionas, of ISZERO does not contain any explicit decision making. Therefore, no logic-significant control flows can be observed, which is one of the major advantages of lambda obfuscation over traditional techniques.

3.2 Data Structures

To implement lambda obfuscation, we need to first design the data structure to represent lambda terms in the C language. Lambda terms belong to one of the three types, which are variable, abstraction and application. We use enum structure to enumerate all three types, namely Tlam, Tapp and Tvar. Because lambda terms are defined inductively, the data structure we use needs to be capable of referring and linking to other lambda terms easily. We define a C struct called term including two main fields, type and data. Type field stores the type of lambda terms. Data field stores different information based on the type of the lambda term: for a variable, it only stores the identifier, which is a char; for an abstraction, it includes a char to store the variable and a term pointer as the function body; for an application, it includes two term pointers to link the two expressions. Table 3.1 shows the data structure terms used in our implementation and Figure 3.1 presents the SUCC operator and Church numeral 2 using our data structure.

The benefit of representing lambda calculus using the *term* data structure is

Type	Data
Tvar	Char var
Tlam	Char var term* body
Tapp	term* left term* right

Table 3.1. C data structure: term

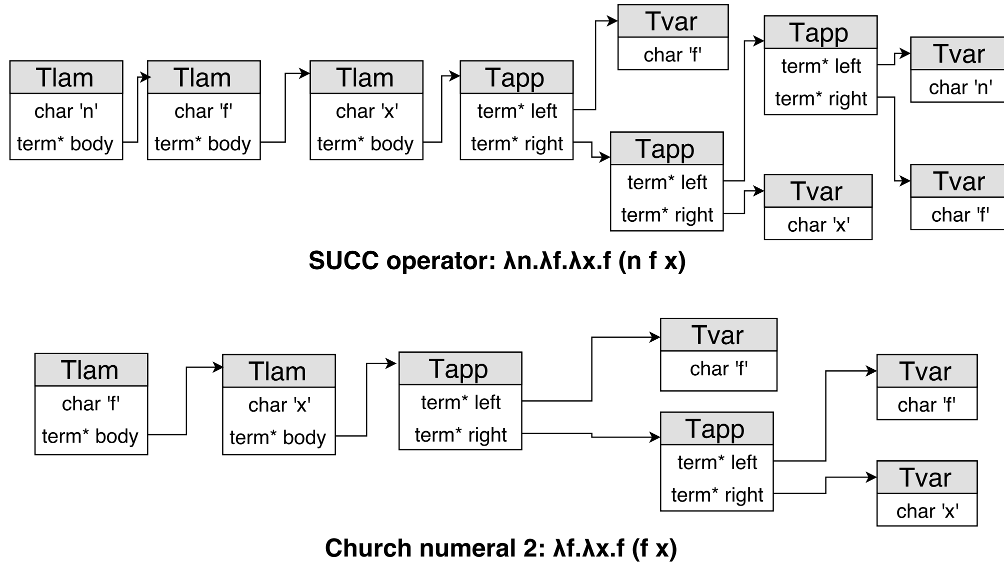

Figure 3.1. SUCC operator and Church numeral 2 in term structure

twofold. First of all, our data structure along with the computation model of lambda calculus makes the execution flow more complicated for analysis tools to make sense. In imperative execution model, computation is conducted through series of explicit instructions that modify memory state [21]. When performing computation in lambda obfuscation, it becomes manipulating *term* objects, such as creating new *term* objects, changing *term* pointers, modifying variable identifier or removing *term* objects. It requires the analysis tools to trace modifications of every intermediate steps to understand the internal logic, which is not only resource-intensive but also time-consuming. Our data structure along with lambda calculus's unique execution model significantly increases the cost and difficulty for

binary analysis tools to make sense the internal logic of obfuscated programs. Secondly, using structure *term* to represent numbers can also make the program more confusing for analysis tools to understand. While Church encoding adopts a significant different approach to encode integer numbers, there is no more clear indications of what integer numbers the function is operating on. Instead, numbers become a link of *term* objects. It increases the cost to trace a value in lambda calculus because it requires adversaries to trace the whole link of *term* objects to identify the number. Moreover, in lambda calculus, every expression can be considered as a function. The Church numerals are simply high-order functions that take functions as arguments and return functions as results. From this point, Church numerals are no different than other lambda calculus operators, such as PLUS operator or SUB operator. During evaluation process, data and operator logic are mixed together and can be difficult to differentiate and make sense. Therefore, leveraging this simple data structure we design to represent lambda calculus in our implementation can make the obfuscated program more obscure for attackers to reverse engineer.

3.3 Lambda Obfuscation

Theoretically, the lambda calculus is able to express all kinds of computation available through modern programming languages, thanks to its Turing completeness. However, obfuscating the entire program is usually against the common software engineering practices due to considerable performance and maintenance cost. Therefore, software developers usually need to manually pick the part of code

they consider sensitive and vulnerable as obfuscation candidates.

To demonstrate the value of work, we particularly pick path conditions as the obfuscation targets. To be specific, we re-implement the computation of path predicates with lambda calculus. In most cases, path conditions are the crux of understanding software behavior and computation logic. By focusing on this part, we are able to evaluate lambda obfuscation without domain-specific knowledge about the software we obfuscate.

Branches are usually implemented through comparison. To obfuscate a path condition instruction, we combine the corresponding lambda comparison operator with the compared parameters also encoded as lambda terms, forming a lambda expression that represents the path condition computation. At run time, the lambda expression is evaluated to a form that cannot be further reduced. This irreducible lambda term, which is the computation result, will be decoded back to the imperative value it represents. Typically, a boolean value will be returned and guide the execution of following branching instructions. In this way, the branch information gets protected by lambda obfuscation and the potential leakage of sensitive information to adversaries is prevented.

Chapter 4

Implementation

We build our obfuscator based on LLVM, a compilation framework that is capable of processing various high-level programming languages, such as C language. As shown in Figure 4.1, there are three stages in our obfuscation work-flow. First of all, we compile all related source code into LLVM intermediate representation (IR) code. Our obfuscator identifies all eligible path condition instruction candidates and transforms such instructions into corresponding function calls to lambda calculus interpreter. The obfuscated IR codes are finally linked and compiled into executable binary in the last stage. We implement our lambda calculus interpreter in C language which consists of total 736 lines of code.

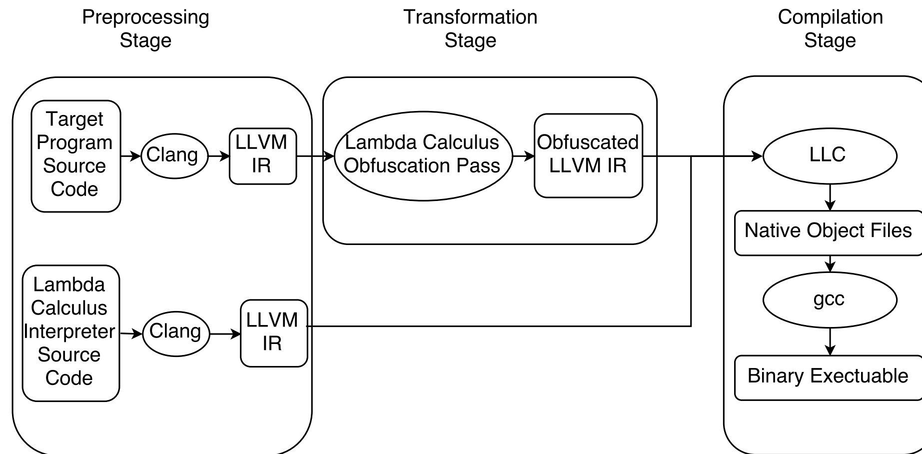

Figure 4.1. The workflow of lambda obfuscation

Source Code

int main() {

...

if (***x + 1 > 1***)

...

Original LLVM IR Code

define i32 @main() #0 {

entry:

...

%0 = load i32, i32* %x, align 4

%add = ***add nsw i32 %0, 1***

%cmp = ***icmp sgt i32 %add, 1***

br i1 %cmp, label %if.then, label %if.end

...

Obfuscated LLVM IR Code

define i32 @main() #0 {

entry:

...

%0 = load i32, i32* %x, align 4

%add = ***call i32 @lambda_add(i32 %0, i32 1)***

%cmp = ***call i1 @lamb_callee(i32 38, i32 %add, i32 1)***

br i1 %cmp, label %if.then, label %if.end

...

Figure 4.2. LLVM IR code of a C program before and after obfuscation

4.1 Stage One: Preprocessing

In the LLVM framework, the majority of program analysis and optimization phases are conducted at LLVM IR level. In order to leverage the strength of LLVM framework, we first compile related source code into LLVM IR code in preprocessing stage. The compilation is conducted without any optimization selections so the IR code captures unmodified behavior of the original program. Input source code comprises not only source code of the program to be obfuscated but also the implementation code of our lambda calculus interpreter. However, only the LLVM IR code generated from the target program source code will be obfuscated in our transformation stage. Because we select C programs to evaluate the effectiveness of our obfuscator, we use Clang as our front-end compiler to generate LLVM IR code during our preprocessing stage.

4.2 Stage Two: Transformation

The strength of LLVM comes from the fact that it provides an easy-to-extend LLVM pass framework. In LLVM, a pass takes LLVM IR code to perform transformation and optimization on the program. Users can customize and implement various passes based on their needs and requirements. We implement a LLVM pass to identify potential path condition instruction candidates and perform obfuscation transformation.

4.2.1 Identifying Instruction Candidates

After preprocessing stage, LLVM IR code that generated from source code will be fed into our LLVM pass for analysis. Every IR instruction is analyzed by our pass to determine whether it meets our obfuscation requirement. In theory, our technique is capable of obfuscating all kinds of computation. To demonstrate the value of our approach, currently our prototype obfuscates path conditions which serve as crucial parts to determine program's control flow logic. As for the types of instruction, our pass selects the following 6 different types of instructions that compute different path conditions: equal, not equal, greater than, greater or equal, less than, less or equal. Besides, our pass picks instruction candidates randomly based on a predefined percentage.

4.2.2 Transforming Instruction

After identifying instruction candidates, our LLVM pass performs obfuscation transformation on these instructions. Path condition instructions are replaced with corresponding lambda calculus function calls to lambda calculus interpreter with proper input parameters, including the type of instruction and both operands. Our lambda calculus interpreter executes the computation of path condition and returns the result to a register which guides the following branching instruction. In this way, the behavior of path condition instruction is preserved while the branch information is protected and obfuscated. Figure 4.2 shows the LLVM code of a example C program before and after our obfuscation.

4.3 Stage Three: Compilation

In stage three, we compile the obfuscated IR code along with the IR code of our lambda calculus interpreter into native object files using llc, a static LLVM compiler that takes LLVM IR and outputs assembly codes. All the native object files are linked and compiled into the final binary executable by gcc. It is worth noting that since we implement our lambda calculus interpreter in C language, an extra runtime environment to execute our lambda calculus interpreter is not required. This implementation decision also increases the stealthy of our obfuscation approach.

Chapter 5

Evaluation

We evaluate lambda obfuscation in four aspects proposed by Collberg et al. [22]: potency, resilience, cost and stealth. Potency measures how complicated and unintelligible the program has become after obfuscation. Resilience indicates how much difficulty the obfuscation can raise for automated reverse engineering techniques. Cost measures how much the software is slowed down as the cost of obfuscation. Stealth describes to what extent the obfuscated program resembles the original program such that the presence of obfuscation is hard to detect.

For the purpose of evaluation, we pick two real-world open source C programs to obfuscate with our lambda obfuscation prototype. The two programs are bzip2, a file compressor, and regexp, a regular expression engine. Both applications contain many integer conditional branches so they can provide enough path condition instruction candidates. The variety of application areas also helps prove that our obfuscator can be applied to different real-world application scenarios.

In our experiments, we obfuscate the two applications at different obfuscation levels, defined as the percentage of qualified path condition instructions that are obfuscated. For example, an application obfuscated at the 20% obfuscation level indicates that the 80% of the original integral path conditions remain unmodified while the rest 20% are transformed into lambda calculus function calls. To avoid biased experiment conditions, we randomly select path conditions to obfuscate. In reality, however, the program components to protect are usually identified by developers with care to achieve the highest possible cost-effectiveness.

5.1 Potency

In order to quantify the potency of lambda obfuscation, we first measure three basic program metrics that are derived from call graphs and control flow graphs before and after transformation. The metrics are the number of edges in the call graph, the number of edges in the control flow graph, and the number of basic blocks. With the help of IDA pro, a widely-used disassembler, we generate call graphs and control graphs from binaries that compiled from original and obfuscated source code.

In addition to these basic metrics, we also calculate two advanced indicators of software complexity which are widely used in the software engineering community, i.e., the cyclomatic number [23] and the knot count [24]. The cyclomatic number is defined as $E - N + 2$ where E is the number of edges and N is the number of vertices in the program's control flow graph. The knot count, on the other hand, is the count of intersections among the control flow paths through a function.

	bzip2	Obfuscated bzip2	regexp	Obfuscated regexp
# of Call Graph Edges	620	1049	144	380
# of Basic Blocks	2590	2839	392	643
# of CFG Edges	3795	4155	562	883
Knot Count	3162	3304	482	616
Cyclomatic Complexity	1207	1278	172	242

Table 5.1. Program metrics before and after obfuscation at obfuscation level 30%

Table 5.1 presents the potency-related statistics of the two evaluated applications before and after obfuscation at obfuscation level 30%. As can be seen through the results, the complexity of both applications has increased by a significant amount after being obfuscated indicating that lambda obfuscation is able to make programs more difficult for attackers to reverse engineer.

5.2 Resilience

For resilience evaluation, we perform concolic testing on an arbitrary C program before and after obfuscation using our approach. Concolic testing is initially a software verification technique combing concrete execution of a program with symbolic execution. Concolic testing aims at covering as many feasible execution paths of a program as possible [25]. However, attackers can use concolic testing to reveal sensitive control flow information of a program and steal valuable program semantics. By performing concolic testing experiment, we try to imitate a reverse engineering attack on programs protected by lambda obfuscation. We select a popular concolic testing tool, KLEE, which is capable of automatically generating test cases and achieving a high coverage of possible execution paths [4]. The subject program to be obfuscated is a simple C program shown in 5.1. We use this extremely simple program so that we can rule out irrelevant factors that will affect the performance

```
1    int test(int a) {
2        int Var = a;
3        if(Var > 16) {
4            Var++;
5        }
6        return Var;
7    }
8
9    int main() {
10       int a;
11       klee_make_symbolic(&a, sizeof(a), "a");
12       return test(a);
13   }
```

Figure 5.1. C program to be obfuscated in KLEE experiment

of KLEE.

We report that KLEE can easily perform concolic testing on the original C program. It succeeds in discovering both paths of the C program and generating test cases for the original program. In contrast, KLEE fails to generate any possible paths for our obfuscated C program. While there are too many possible states for KLEE to explore and reason, KLEE keeps hitting the maximum memory capacity and eventually stops without returning any possible paths. This result indicates that lambda obufscation makes an extremely simple program so complicated that KLEE is no longer capable of revealing any useful control flow information of the protected program.

5.3 Cost

The main overhead introduced by lambda obfuscation mainly comes from the encoding and decoding translation process and the reduction time of lambda calculus. In order to measure the cost of our technique, we apply obfuscation to bzip2 and

Program	before	after	executed times	overhead
bzip2	0.0625s	15.574s	375,351	41.492μs
regexp	0.413s	28.716s	822,873	34.89μs

Table 5.2. Overhead of lambda calculus obfuscation on bzip2 and regexp

regexp at the obfuscation level of 30%, meaning 30% of the path conditions are obfuscated. The test inputs we use for the experiments are shipped with the original source code. Each application is ran 10 times and the average run time is presented with the slowdown.

Table 5.2 compares the execution time of both applications before and after obfuscation. We also record how many times is our lambda calculus interpreter invoked during each application's runtime and we calculate the average overhead. As Table 3 shown, on average every single call to our lambda calculus interpreter requires 38.19 μs. We believe the cost is moderate and comparable to normal function calls. Besides, we argue that the overhead of our lambda calculus obfuscation is reasonable and can be reduced. Since we choose our path condition instruction candidates totally at random, some of the obfuscated path condition instructions reside in hot spots, which means these path condition instructions are being intensively called and used during runtime. For example, some of the path condition instructions we obfuscated in bzip2 reside in for loops which eventually accumulate to slowdown the program. In such cases, the overhead introduced by lambda obfuscation is inevitable and forgivable. In practice, users can obfuscate path conditions that are sensitive while less intensively-used to gain the maximum benefit from our obfuscation technique.

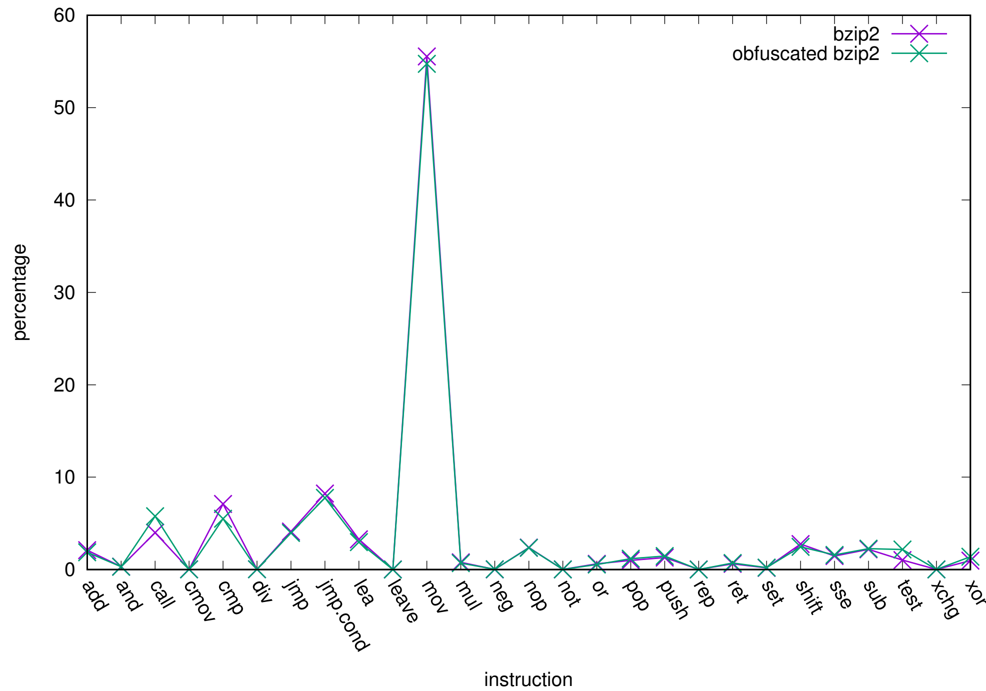

Figure 5.2. Instruction distribution of bzip2

5.4 Stealth

To measure how stealthy our obfuscation technique can be to avoid detection, we collect instruction distribution of our obfuscated sample C programs and compare them with the distribution of their original source code.

Figure 5.2 and Figure 5.3 show the instruction distribution of the original and obfuscated programs at obfuscation level of 30%. As we can see from the figures, the distribution of our obfuscated programs is very similar to their original distributions. In this case, we believe that the behavior of our obfuscated programs resembles their original behavior and the existence of our obfuscation can avoid being detected by adversaries.

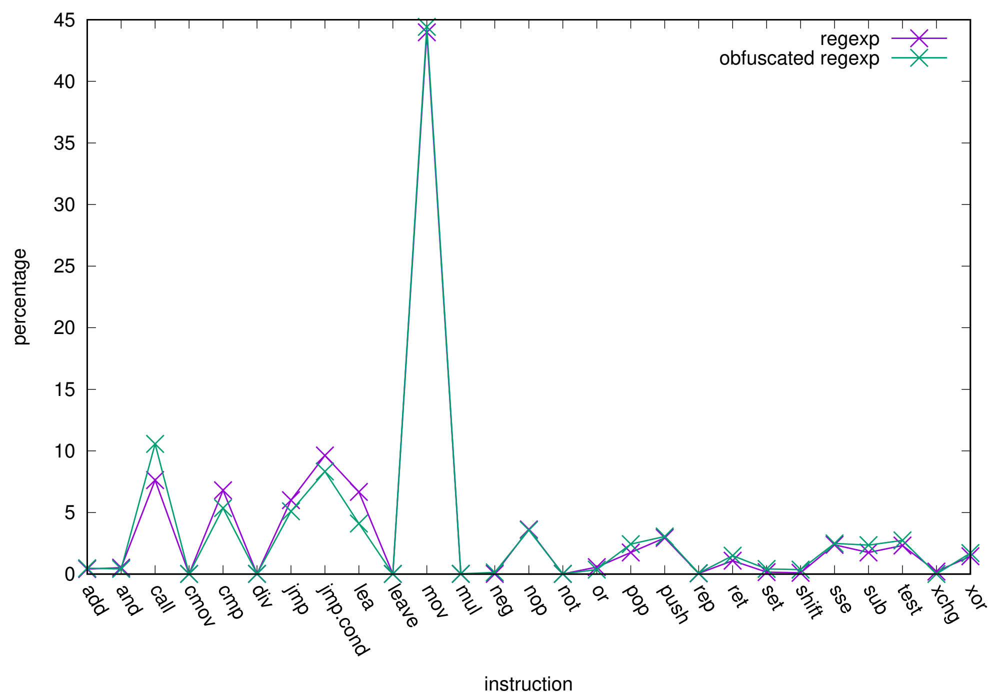

Figure 5.3. Instruction distribution of regexp

Chapter 6

Discussion

6.1 Countering Dynamic Monitoring

Opaque predicates and many other control flow obfuscation methods are inherently vulnerable to dynamic analysis, i.e., attackers monitoring the execution of the obfuscated software and checking control flows at run time. Lambda obfuscation may face similar challenges when only partially applied to protecting branch conditions. Learning from previous work, we found that there are several ways to alleviate this issue. One possible countermeasure is to blur the boundaries between lambda simulation and the original program code, using heuristics like function inlining and jumps across functions [26].

6.2 Potential Extensions

Currently, our lambda calculus obfuscator is applied to path condition instructions that involve integer calculation. The limitation of our technique is caused by the fact that Church encoding is only capable of encoding natural number instead of real number. According to Church-Turing thesis, any data types can be encoded using lambda calculus [20]. One way to encode real number is using a Cauchy sequence of rational numbers [27]. After properly encoding real number in lambda calculus, we can extend our approach to obfuscate instructions involving real number.

Lambda calculus can also be extended to obfuscate other instructions besides path condition instructions that we currently focus on. Lambda calculus interpreter is capable of handling multiple arithmetic operations, such as addition, subtraction. Our obfuscator can be applied to any instructions containing such operations. In order to obfuscate these instructions, we can extend our LLVM obfuscator to identify suitable instructions and replace them with corresponding lambda calculus function calls.

Another way to enhance the obfuscating effect is to implement indirect control transferring similar to the obfuscation schema proposed by Ma et al. [14]. Currently, our obfuscator replaces path condition instructions with lambda function calls that return boolean signals to guide following conditional jump instructions. Instead of returning boolean signals, the obfuscator can return instruction addresses and we can modify following conditional jump instructions to be unconditional jump instructions that take instruction addresses. In this way, we

can transform conditional logic into unconditional control transfer to make the obfuscated programs even more confusing for attackers to make sense.

6.3 Combining with Other Obfuscation

In this thesis, we implement the lambda transformation at LLVM IR level using pass framework. In LLVM, every pass can be considered as an independent optimization of the original program and multiple different passes can be applied if needed. Therefore, lambda calculus is compatible with other obfuscation techniques if they happen at source code level or at LLVM IR level. Lambda calculus obfuscation can served as an extra obfuscation layer to be applied before compilation of the program with other obfuscation techniques to make the program more obscure and secure. Besides, lambda obfuscator comes with reduction rules to evaluate lambda calculus which means the obfuscated program can run without an extra runtime environment. It can independently encode and decode lambda numerals and perform the whole evaluation process. This independent characteristic makes lambda calculus obfuscation less possible to affect other obfuscation techniques if applied together.

6.4 Obfuscating Complete Branch Predicates

Currently, in the obfuscated program, path condition instructions are replaced with lambda calculus function calls with instruction type and operands as input

parameters. In order to further limit adversaries' knowledge to program semantic, we can further obfuscate instruction information. One possible solution is to encode all instruction information using lambda calculus and combine them into one single lambda term. Every instruction can be transformed into a different lambda calculus function which encapsulates the lambda calculus term that represents the instruction type and operands. By calling every instruction-specific function, our lambda calculus evaluator can still simulate the behavior of each obfuscated instruction. In this way, the instruction information is concealed through lambda calculus encoding and less program semantic is leaked to attackers.

Chapter 7

Conclusion

In this thesis, we propose a novel obfuscation technique based on lambda calculus. The behavior of path condition instructions is simulated using lambda calculus while sensitive instruction information is concealed. The complicated execution model of lambda calculus makes the obfuscated programs more obscure for the adversaries to make sense and reverse engineer. We implement a lambda obfuscator that transforms path condition instructions into corresponding lambda calculus function calls. A lambda interpreter is also implemented to evaluate lambda calculus function calls and return boolean signals to guarantee the behavior of original path condition instructions is still preserved. We evaluate our prototypical implementation of lambda obfuscation with respect to potency, resilience, cost and stealthy. Our experiment result shows that our obfuscation technique can make the program more obscure with moderate overhead.

Bibliography

[1] Viticchié, A., L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella (2016) "Assessment of Source Code Obfuscation Techniques," in *Proceedings of 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*, pp. 11–20.

[2] Rojas, R. (2015) "A Tutorial Introduction to the Lambda Calculus," *CoRR*, **abs/1503.09060**.

[3] Lattner, C. and V. Adve (2004) "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, pp. 75–86.

[4] Cadar, C., D. Dunbar, D. R. Engler, et al. (2008) "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pp. 209–224.

[5] Balachandran, V. and S. Emmanuel (2013) "Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code," *IEEE Transactions on Information Forensics and Security*, **8**(4), pp. 669–681.

[6] Xu, D., J. Ming, and D. Wu (2016) "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method," in *Proceedings of the 19th Information Security Conference (ISC'16)*, pp. 323–342.

[7] Myles, G. and C. Collberg (2006) "Software Watermarking via Opaque Predicates: Implementation, Analysis, and Attacks," *Electronic Commerce Research*, **6**(2), pp. 155–171.

[8] Arboit, G. (2002) "A Method for Watermarking Java Programs via Opaque Predicates," in *Proceedings of The Fifth International Conference on Electronic Commerce Research (ICECR'02)*, pp. 102–110.

[9] Palsberg, J., S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang (2000) "Experience with Software Watermarking," in *Proceedings of 16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 308–316.

[10] Sharif, M. I., A. Lanzi, J. T. Giffin, and W. Lee (2008) "Impeding Malware Analysis Using Conditional Code Obfuscation," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[11] Falcarin, P., S. Di Carlo, A. Cabutto, N. Garazzino, and D. Barberis (2011) "Exploiting Code Mobility for Dynamic Binary Obfuscation," in *Proceedings of 2011 World Congress on Internet Security (WorldCIS'11)*, pp. 114–120.

[12] Wang, Z., C. Jia, M. Liu, and X. Yu (2012) "Branch Obfuscation Using Code Mobility and Signal," in *Proceedings of 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW'12)*, pp. 553–558.

[13] Rauti, S., S. Laurén, S. Hosseinzadeh, J.-M. Mäkelä, S. Hyrynsalmi, and V. Leppänen (2014) "Diversification of System Calls in Linux Binaries," in *Revised Selected Papers of the 6th International Conference on Trusted Systems (INTRUST'14)*, pp. 15–35.

[14] Ma, H., X. Ma, W. Liu, Z. Huang, D. Gao, and C. Jia (2014) "Control Flow Obfuscation Using Neural Network to Fight Concolic Testing," in *Proceedings of 10th International Conference on Security and Privacy in Communication Networks (SECURECOMM'14)*, pp. 287–304.

[15] Wang, P., S. Wang, J. Ming, Y. Jiang, and D. Wu (2016) "Translingual Obfuscation," in *Proceedings of 2016 IEEE European Symposium on Security and Privacy (EuroS&P'16)*, pp. 128–144.

[16] Pierce, B. C. (2002) *Types and Programming Languages*, MIT Press.

[17] Hudak, P. (1989) "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys (CSUR)*, **21**(3), pp. 359–411.

[18] De Queiroz, R. J. (1988) "A Proof-Theoretic Account of Programming and the Role of Reduction Rules," *Dialectica*, **42**(4), pp. 265–282.

[19] Slonneger, K. and B. L. Kurtz (1995) *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Longman Publishing Co., Inc.

[20] García Pérez, Á. (2014) *Operational Aspects of Full Reduction in Lambda Calculi*, Ph.D. thesis, E.T.S. de Ingenieros Informticos (UPM).

[21] Chailloux, E., P. Manoury, and B. Pagano (2002) *Developing Applications with Objective Caml*, O'Reilly France.

[22] Collberg, C., C. Thomborson, and D. Low (1998) "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pp. 184–196.

[23] McCabe, T. J. (1976) "A Complexity Measure," *IEEE Transactions on Software Engineering*, **SE-2**(4), pp. 308–320.

[24] Woodward, M. R., M. A. Hennell, and D. Hedley (1979) "A Measure of Control Flow Complexity in Program Text," *IEEE Transactions on Software Engineering*, **5**(1), pp. 45–50.

[25] Giantsios, A., N. Papaspyrou, and K. Sagonas (2015) "Concolic Testing for Functional Languages," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, pp. 137–148.

[26] Ma, H., R. Li, X. Yu, C. Jia, and D. Gao (2016) "Integrated Software Fingerprinting via Neural-Network-Based Control Flow Obfuscation," *IEEE Transactions on Information Forensics and Security*, **11**(10), pp. 2322–2337.

[27] Geuvers, H., M. Niqui, B. Spitters, and F. Wiedijk (2007) "Constructive Analysis, Types and Exact Real Numbers," *Mathematical Structures in Computer Science*, **17**(1), pp. 3–36.

Publications

[1] Jiang Ming, Zhi Xin, Pengwei Lan, Dinghao Wu, Peng Liu, and Bing Mao. "Impeding Behavior-based Malware Analysis via Replacement Attacks to Malware Specications," *Journal of Computer Virology and Hacking Techniques*, 2016.

[2] Jiang Ming, Zhi Xin, Pengwei Lan, Dinghao Wu, Peng Liu, and Bing Mao. "Replacement Attacks: Automatically Impeding Behavior-based Malware Specifications," in *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS 2015)*, New York, June 2–5, 2015.