Lambda Obfuscation

Pengwei Lan, Pei Wang, Shuai Wang, and Dinghao Wu

College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{pul139,pxw172,szw175,dwu}@ist.psu.edu

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 paper, 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 function calls 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.

Key words: Software obfuscation, control flow obfuscation, reverse engineering, lambda calculus

1 Introduction

As binary analysis techniques keep advancing, reverse engineering is becoming more effective than ever before. Consequently, malicious parties are able to employ the latest binary analysis techniques to identify exploitable software vulnerabilities for injecting malicious code 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 are program transformations that make software more complicated than its original form and difficult for adversaries to understand and analyze, while preserving the program's original semantics [24].

In this paper, we propose a novel obfuscation method, called lambda obfuscation, that 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 security-sensitive parts of the original programs. Instead of imperatively performing computation with data and control step by step, 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 integer, are represented as high-order functions, potentially making conventional information flows implicit. When this highly abstract computation model is implemented and deployed with low-level machine code, a huge semantics gap emerges and imposes great challenges on manual and automated program analysis, therefore hindering reverse engineering.

Being Turing complete and considered as the smallest universal programming language [21], lambda calculus is capable of expressing all kinds of computation patterns available with a typical imperative programming language, e.g., C, Pascal, and Fortran. If the simulated computation is free of side effects, the source-level conversion can be fairly straightforward, yet the resulting program binary after transformation will become much more complicated and obscure.

To demonstrate the feasibility and practicality of lambda obfuscation, we implemented a prototypical lambda obfuscator based on the LLVM compiler infrastructure [13]. The obfuscator transforms qualified branch conditions into lambda calculus terms that simulate their original behavior. In order to return the simulation results, an interpreter that evaluates the lambda calculus is linked to compiled binaries including the procedures and intermediate values for computing the heavily obfuscated results. We comprehensively evaluated our obfuscation technique in four aspects, namely potency, resilience, cost, and stealth. The evaluation result indicates 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 [3] and obtained promising results.

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

2 Related Work

Software obfuscation techniques can be divided into four major categories, namely layout obfuscation, preventive obfuscation, data obfuscation, and control obfuscation [2]. Arguably as the most popular one, control obfuscation focuses on concealing and complicating control flow information of the program. There has been a large volume of research striving to develop effective control obfuscation techniques from different angles.

One of the classic approaches to achieving control obfuscation is by designing resilient opaque predicates. A predicate is opaque if it evaluates to a predetermined constant regardless of its input, while this invariant is hard to reveal through static analysis [30]. Most opaque predicates are derived from number-theoretic theorems [17], e.g., the quadratic residue lemmas [1]. One of the fundamental drawbacks of employing opaque predicates is that they always evaluate to the same value at run time, thus vulnerable to dynamic analysis. 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. [18] introduced dynamic opaque predicates in which a family of correlated predicates whose evaluation results are only invariant in specific execution contexts.

Sharif et al. [22] proposed a conditional code obfuscation technique that leverages the inconvertibility of cryptographic hash functions to protect branch conditions. They used the hash functions to obfuscate the value of variable for which the branch condition can be satisfied. Because of the preimage resistance properties of these cryptographic hash functions, it is not practically feasible for static 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 applicable to branch conditions evaluated through the equality relation, while it fails to protect conditions that contain inequality relations.

There is a line of research on building obfuscation techniques based on code mobility [7, 28, 20]. These 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 heavily rely on the availability of network communications and remote servers, which limits the application scenarios of their techniques.

Control flow obfuscation can also be implemented by introducing exotic computation gadgets and paradigms. Ma et al. [15] 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. [25] introduced another obfuscation framework called translingual obfuscation. They proposed to translate programs written in imperative programming languages, which are relatively easier to reverse engineer, to languages of different paradigms. In particular, they demonstrated the feasibility of obfuscating C programs with Prolog, a logic programming language based on first-order logic resolution. Due to the vastly different execution models of the original and target languages, traditional binary analysis methods have difficulty in countering translingual obfuscation. Another obfuscation method, called Turing obfuscation [27], augmented the concept of translingual obfuscation by transforming C programs into compositions of primitive Turing machines rather than programs written in another language. Our research shares a similar idea with Turing obfuscation, while we adopt lambda calculus as the foundation of obscurity, which is a more heterogeneous computation model.

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. Even though different programming languages adopt different execution models, it is considered relatively easier to reverse engineer imperative languages whose computation schemas align the best with the underlying hardware. 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 increase the difficulty of de-obfuscation. Therefore, we can translate and implement functionalities of a program using different programming languages to mix execution models and conceal sensitive program information. In this paper, our lambda obfuscation technique 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 computation [19]. 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:

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

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 the variable x and the function body is another lambda term e. An application term captures the action of applying a function to its arguments. For example, lambda expression $(f\ t)$ means applying function f to an expression t, which is provided as the argument to f. All valid lambda terms can be formed by repeatedly combining the three basic lambda terms. Below are some examples of valid lambda terms:

x	A variable x
$\lambda x.x$	An identity function
$(\lambda x.x)\ y$	Applying identity function to variable y
$\lambda p.\lambda q.p\ q$	A function applying its first argument to the second one

When the λ symbol precedes a variable, it binds all the occurrences of this variable in the abstraction body. A variable is called a *bound variable* if its name is associated with a λ symbol. Other variables in the function body are called *free variables* [11]. For example, in the following expression, variable x is a bound variable while variable y is a free variable.

$$\lambda x.x\ y$$

Reduction The meaning of lambda calculus is defined by how lambda calculus can be reduced [6]. 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 [23]. 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 $e_1[x \rightarrow e_2]$ denotes substituting all free occurrences of the variable x with e_2 in 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.

$$\begin{aligned}(\lambda x.x)\ y &\Rightarrow y\\ (\lambda x.x)(\lambda y.y) &\Rightarrow \lambda y.y\\ (\lambda x.x\ x)(\lambda y.y) &\Rightarrow (\lambda y.y)(\lambda y.y) \Rightarrow \lambda y.y\end{aligned}$$

3.2 Church Encoding

In lambda calculus, abstracts, or functions, is the only primitive type that is naturally available. Therefore, to perform meaningful computation that resembles what a modern programming language is capable of, it is imperative to find an encoding scheme to express basic data types like integer and operators in lambda calculus. For the purpose of lambda obfuscation, we employ Church encoding to represent natural numbers and operators to implement lambda obfuscation. In this section, we briefly introduce the basics of Church encoding.

Firstly developed by Alonzo Church, Church encoding describes the value of a natural number as the number of times for which a function is applied to an argument. Natural numbers expressed this way are called Church numerals. For example, when encoded as a Church numeral, the natural number 2 is a lambda abstraction that applies its first argument to its second argument twice. The Church numerals can be defined as follows:

$$\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f.\lambda x.f\ x \\
2 &\equiv \lambda f.\lambda x.f\ (f\ x) \\
3 &\equiv \lambda f.\lambda x.f\ (f\ (f\ x)) \\
n &\equiv \lambda f.\lambda x.f^n\ x
\end{aligned}$$

As the definition indicates, the 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:

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

Within this context, the addition operator can be accordingly defined as a lambda expression. Conceptually, adding m to n is equivalent to adding 1 to n for m times. Therefore, 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:

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

Note that the PRED and SUB operators defined above are “truncated”, meaning the decrementation stops at 0. This is expected since we have not defined negative integers yet, which it is entirely feasible in lambda calculus. Due to limited space, we do not list the complete definitions of all primitives employed lambda obfuscation. If interested, readers can find the corresponding information in many materials on programing languages and logic[1].

Through implementing the Church encoding of necessary operators, we are able to perform basic arithmetic operations in lambda calculus, including addition, subtraction, multiplication, and division. We can also simulate equality and all kinds of inequality comparisons, e.g., greater than, smaller than. 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 terms with which we can simulate the computation of path conditions in typical C programs.

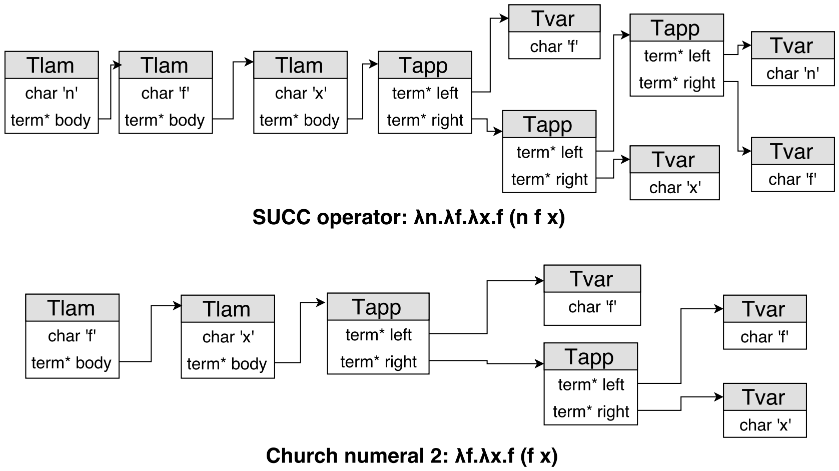
From the perspective of program obfuscation, the Church encoding "accidentally" possesses the capability of eliminating explicit control flows. As an example, the ISZERO lambda term simulates a typical branch operation in imperative programming. However, the computation, or more precisely the reduction, 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.

3.3 Data Structures

To implement lambda obfuscation, we need to first design the data structure to represent lambda terms. As introduced earlier, a lambda term can be one of the variable type, abstraction type, and application type. Naturally, we use enum structure to enumerate all three types, namely `Tvar`, `Tlam`, and `Tapp`. Because lambda terms are defined inductively, the data structure we use needs to refer and link to other lambda terms recursively. We define a C struct called `term` including two main fields, i.e., type and data. The type field stores the type of lambda terms. The 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 the abstraction structure, it includes a char to store the variable and a term pointer as the function body. An application consists of two term pointers to link the two expressions. Figure 1 presents the SUCC operator and Church numeral 2 using the data structures described above.

The benefit of representing lambda calculus with the `term` data structure is twofold. Firstly, our data structure, along with the computation model of lambda calculus, makes the execution flow more complicated for analysis tools to reason about. In the imperative execution model, computation is conducted through series of explicit instructions that modify memory states [4]. While in lambda

[1] Our current implementation does not support floating point numbers and arithmetic, but it is feasible and can be added into the implementation with more engineering effort.



SUCC operator: λn.λf.λx.f (n f x)

Church numeral 2: λf.λx.f (f x)

Fig. 1. SUCC operator and Church numeral 2 in term structure

obfuscation, computation is conducted through manipulating `term` objects, such as creating new `term` objects, changing term pointers, modifying variable identifier, and removing existing `term` objects. Thus, it requires analysis tools to trace the modifications of every intermediate steps to understand the internal logic which is not only resource-intensive but also time-consuming. Our data structure and unique execution model and lambda calculus significantly increase the cost and difficulty for binary analysis tools to reveal the internal logic of obfuscated programs. Secondly, the Church encoding, and our implementation of it using `term`, is "unnatural" by itself in the first place. The encoding adopts a significant different approach to encode natural numbers and other data types that are mostly primitive in a traditional imperative computation model. Instead, numbers become a link of `term` objects. As such, there are no more clear indications on what numbers the computation is operating on. This notably increases the cost to trace a value in lambda calculus because it now requires adversaries to trace the whole link of `term` objects to identify the number. Moreover, with Church encoding, every expression can is represented as a function, making data and operation much less distinguishable. In particular, 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 the evaluation process, data and operator logic are mixed together. In summary, leveraging this simple data structure we design to represent lambda calculus in our implementation can make the obfuscated programs more obscure for attackers to reverse engineer.

3.4 Lambda Obfuscation

Theoretically, the lambda calculus is mostly as powerful as a modern programming language, due 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 have to manually pick the part of code they consider sensitive and vulnerable as obfuscation candidates.

To demonstrate the value of lambda obfuscation, we particularly pick path conditions as the targets to apply obfuscation to. To be specific, we re-implement the computation of path predicates with lambda calculus. Path conditions, in most software, are the crux of understanding program 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 which are both 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, namely the computation result, will be decoded back to the imperative value it represents. Typically, a boolean value will be returned to guide the execution of following branching instruction. In this way, the branch information gets protected by lambda obfuscation and many potential leakages of sensitive information to adversaries can be prevented.

4 Implementation

We implement lambda obfuscation based on LLVM, a architecture-independent compilation framework supporting flexible program transformations. As shown in Figure 2, our obfuscation work-flow is divided into three stages. The first step is preprocessing. In this stage, we compile all source code to be obfuscated into the LLVM intermediate representation (IR). The next step is transformation, in which the obfuscator identifies all eligible instructions used for path condition computation and translate these instructions into lambda calculus terms. These instructions will then be replaced by trampolines to a lambda calculus interpreter that accepts the generated lambda calculus terms as input. In the last compilation stage, the obfuscated IR code are compiled to machine code and linked into an executable binary. The lambda calculus interpreter is implemented with 736 lines of C code [2]. We elaborate on the details of each stage below.

[2] For more implementation details, please refer to an extended version of this paper [12]

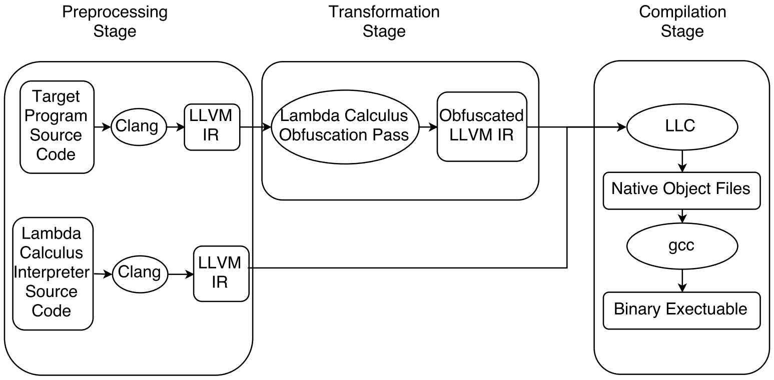
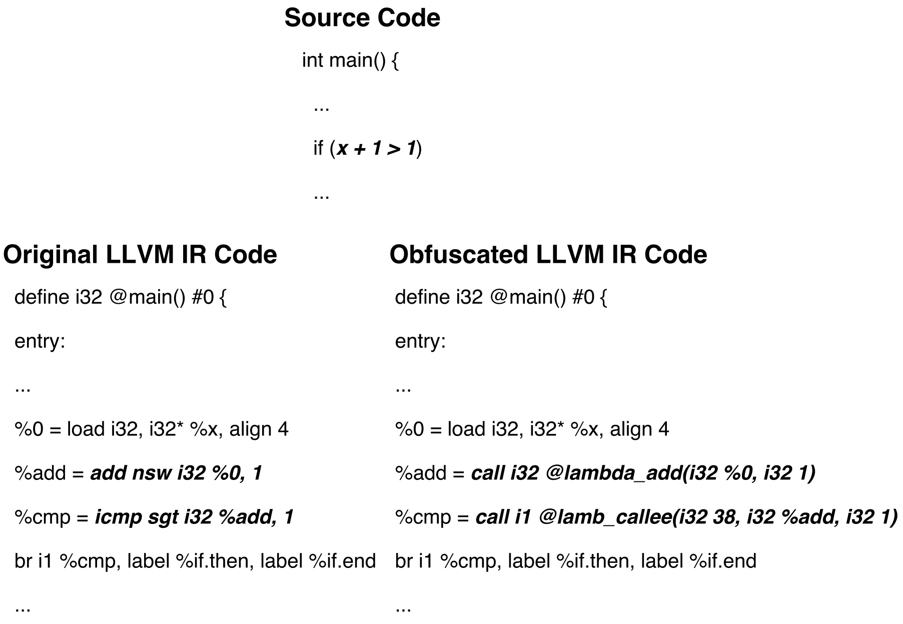


Fig. 2. The work-flow of lambda obfuscation



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
...
```

Fig. 3. LLVM code of a C program before and after obfuscation

4.1 Preprocessing

In LLVM, the majority of program analysis and optimization phases are conducted at the LLVM IR level. In order to leverage the strength of the transformation framework, we compile the source code into LLVM IR code. The compilation is conducted without any optimization so the IR code captures the unmodified behavior of the original program. The input source code comprises not only source code of the program to be obfuscated but also the implementation of our lambda calculus interpreter. However, only the LLVM IR code generated from the target program source code will be obfuscated in the 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 Transformation

LLVM provides an easy-to-extend pass-based transformation framework. Users can customize and implement passes at different program level based on their needs and requirements. We implement a function pass that processes each function in a compile unit to identify instructions that are suitable for obfuscation.

Identifying Instruction Candidates After the preprocessing stage, LLVM IR code generated from the source is fed into another LLVM pass for analysis. Every IR instruction is analyzed to determine whether it meets our obfuscation requirement. In theory, lambda obfuscation is capable of obfuscating all kinds of computation. At this point, our prototype obfuscates path conditions which serve as crucial parts forming the control flows of a program. As for the types of instruction, the pass selects the following six types of instructions that compute different path conditions: equal, not equal, greater than, greater or equal, less than, and less or equal. To allow users to control the strength of obfuscation, the pass picks instruction candidates randomly based on a percentage specified by users.

Transforming Instructions After identifying the candidates for obfuscation, a translation pass performs lambda transformation for these instructions. Path conditions are replaced by the corresponding lambda calculus function calls to lambda calculus interpreter with proper input parameters, including the type of comparison operators and operands. The lambda calculus interpreter simulates the computation of the path condition and returns the result to a register which is send back to the original program as the computed path condition. Figure 3 shows the LLVM IR code of a example C program before and after our obfuscation.

4.3 Compilation

In the final stage, we compile the obfuscated IR code and the IR code of our lambda calculus interpreter into native machine instructions. It is worth noting that since we implemented the lambda calculus interpreter in C, no additional runtime environment is required to execute the obfuscated binary. This implementation decision also increases the stealthy of our obfuscation approach.

5 Evaluation

We evaluate lambda obfuscation in four aspects, i.e., potency, resilience, cost, and stealth, which are firstly proposed by Collberg et al. [5]. Potency measures how complicated and unintelligible the program has become after obfuscation. Resilience indicates how well the obfuscated program can withstand automated reverse engineering. 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 picked two open source C programs to obfuscate using our lambda obfuscation prototype. The two programs are bzip2, a file compressor, and regexp, a regular expression engine. Both applications contain many integral path conditions therefore enough obfuscation candidates.

In the evaluation, the obfuscation strength is described by a metric called obfuscation level, which is defined as the percentage of obfuscated path conditions with respect to all qualified obfuscation candidates. 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 terms. To avoid being biased in the experiments, 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 measured three basic software complexity 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 disassembler widely used in the industry, we generated call graphs and control graphs from binaries compiled from original and obfuscated LLVM IR code.

In addition to these basic metrics, we also calculated two advanced indicators of software complexity which have long been utilized by the software engineering community, i.e., the cyclomatic number [16] and the knot count [29]. 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 when all basic blocks in the function are linearly aligned.

Table 1 presents the potency-related statistics of the two evaluated applications before and after obfuscation, at the obfuscation level of 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.

Table 1. Program metrics before and after obfuscation at obfuscation level 30%

	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

5.2 Resilience

For resilience evaluation, we performed 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 to cover as many feasible execution paths of a program as possible [10]. However, attackers can use concolic testing to reveal sensitive control flow information of a program and learn about program semantics. By performing concolic testing experiment, we tried to imitate a reverse engineering attack on programs protected by lambda obfuscation. We picked a popular concolic testing tool, KLEE, which is capable of automatically generating test cases and achieving a high coverage of possible execution paths [3]. The program used for testing the matchup between KLEE and lambda obfuscation is the obfuscated binary of a simple C program shown in Figure 4. We used this extremely simple program to rule out irrelevant factors that can possibly affect the performance of KLEE.

```c
int test(int a) {
    int Var = a;
    if(Var > 16) {
        Var++;
    }
    return Var;
}

int main() {
    int a;
    klee_make_symbolic(&a, sizeof(a), "a");
    return test(a);
}
```

Fig. 4. C program to be obfuscated in KLEE experiment

With the experiment, we found that KLEE could successfully finish concolic testing on the unobfuscated binary. To be specific, KLEE succeeded in discovering both paths of the C program and generating test cases for the original program. In contrast, KLEE failed to generate any possible paths for the obfuscated binary. The topmost issue that caused the failure was that there were too many possible states for KLEE to explore and reason such that KLEE kept hitting the maximum memory capacity and eventually stopped without returning any possible paths. This result indicates that lambda obfuscation makes an extremely simple program so complicated that KLEE can no longer reveal any useful control flow information of the protected program.

5.3 Cost

The major source of performance overhead introduced by lambda obfuscation 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 applied obfuscation to bzip2 and regexp at the obfuscation level of 30%. The test input used for the experiments are the original test cases shipped with the source code. Each application was executed 10 times and the average run time is presented with the slowdown.

Table 2. Overhead of lambda obfuscation on bzip2 and regexp

	Interpreter Invocations	**Average Time (Original)**	**Average Time (Obfuscated)**	**Overhead**
bzip2	375,351	0.0625s	15.574s	41.492μs
regexp	822,873	0.413s	28.716s	34.89μs

Table 2 compares the execution time of both applications before and after obfuscation. We also recorded how many times was our lambda calculus interpreter invoked during each application's runtime and we calculated the average overhead. As Table 2 shown, on average every single call to our lambda calculus API 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 chose our path condition instruction candidates totally at random, some of the obfuscated path condition instructions resided in hot spots and these path condition instructions were being intensively called and used during runtime. For example, some of the path condition instructions we obfuscated in bzip2 resided in for loops which eventually accumulated 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 lambda obfuscation.

5.4 Stealth

To measure how stealthy lambda obfuscation is, we collected the distribution of instructions in the obfuscated sample C programs and compared them with that of the original binary.

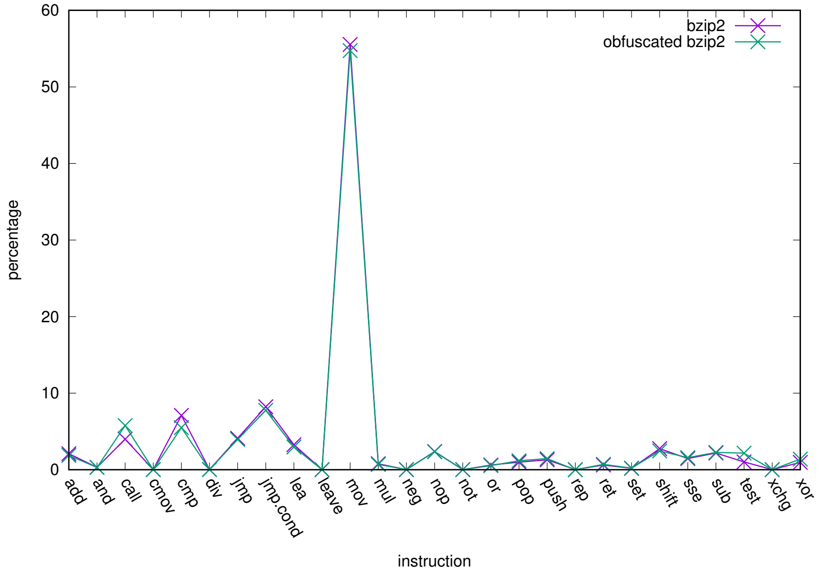


Fig. 5. Instruction distribution of bzip2

Figure 5 and Figure 6 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 the obfuscated programs resembles their original one and it would be very difficult for adversaries to detect the presence of lambda obfuscation through the statistical features of the protected binaries.

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 find that there are several

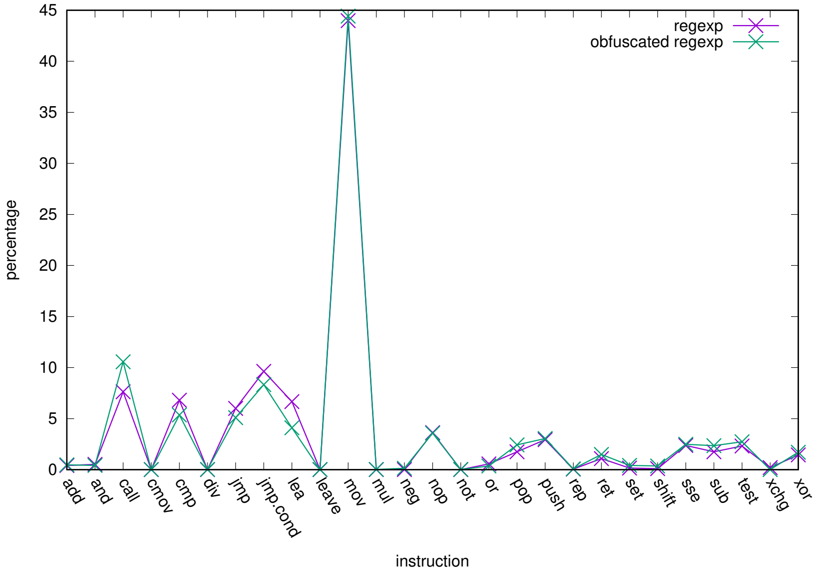


Fig. 6. Instruction distribution of regexp

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 [14].

6.2 Potential Extensions

Currently, lambda obfuscation is only applied to the computation of integral path conditions. 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 [8]. One way to encode real number is using a Cauchy sequence of rational numbers [9]. 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. [15]. 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.

We are also envisioning that obfuscating effect can be notably enhanced by "recursively" applying the proposed technique. That means, we first obfuscate the input program with our Lambda obfuscator, and further re-obfuscate the first round product with our technique. As discussed by existing research [26], such recursive process can even be launched for hundreds of iterations, which could largely increase the program complexity to defeat adversary analysis.

6.3 Combining with Other Obfuscation Methods

In this paper, 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 serve 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.

7 Conclusion

In this paper, we propose a novel obfuscation technique based on lambda calculus. The behavior of path condition instruction 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. The experiment result shows that our obfuscation technique can make the program more obscure with only modest overhead.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912.

References

1. Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *Proceedings of The Fifth International Conference on Electronic Commerce Research (ICECR'02)*, pages 102–110, 2002.
2. Vivek Balachandran and Sabu Emmanuel. Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEEE Transactions on Information Forensics and Security*, 8(4):669–681, April 2013.
3. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 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)*, pages 209–224, 2008.
4. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly France, 2002.
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 (POPL'98)*, pages 184–196, 1998.
6. Ruy JGB De Queiroz. A proof-theoretic account of programming and the role of reduction rules. *Dialectica*, 42(4):265–282, 1988.
7. Paolo Falcarin, Stefano Di Carlo, Alessandro Cabutto, Nicola Garazzino, and Davide Barberis. Exploiting code mobility for dynamic binary obfuscation. In *Proceedings of 2011 World Congress on Internet Security (WorldCIS'11)*, pages 114–120, 2011.

8. Álvaro García Pérez. *Operational Aspects of Full Reduction in Lambda Calculi*. PhD thesis, E.T.S. de Ingenieros Informticos (UPM), 2014.
9. Herman Geuvers, Milad Niqui, Bas Spitters, and Freek Wiedijk. Constructive analysis, types and exact real numbers. *Mathematical Structures in Computer Science*, 17(1):3–36, 2007.
10. Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Concolic testing for functional languages. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, pages 137–148, 2015.
11. Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.
12. Pengwei Lan. Lambda obfuscation. Master's thesis, The Pennsylvania State University, 2017.
13. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, March 2004.
14. Haoyu Ma, Ruiqi Li, Xiaoxu Yu, Chunfu Jia, and Debin Gao. Integrated software fingerprinting via neural-network-based control flow obfuscation. *IEEE Transactions on Information Forensics and Security*, 11(10):2322–2337, 2016.
15. Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. 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)*, pages 287–304, 2014.
16. Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
17. Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.
18. Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. Experience with software watermarking. In *Proceedings of 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 308–316, 2000.
19. Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002.
20. Sampsa Rauti, Samuel Laurén, Shohreh Hosseinzadeh, Jari-Matti Mäkelä, Sami Hyrynsalmi, and Ville Leppänen. Diversification of system calls in linux binaries. In *Revised Selected Papers of the 6th International Conference on Trusted Systems (INTRUST'14)*, pages 15–35, 2014.
21. Raúl Rojas. A tutorial introduction to the lambda calculus. *CoRR*, abs/1503.09060, 2015.
22. Monirul I. 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 (NDSS'08)*, 2008.
23. Kenneth Slonneger and Barry L Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc., 1995.
24. Alessio Viticchié, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella, and Roberto Tiella. Assessment of source code obfuscation techniques. In *Proceedings of 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*, pages 11–20, Oct 2016.
25. Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual obfuscation. In *Proceedings of 2016 IEEE European Symposium on Security and Privacy (EuroS&P'16)*, pages 128–144, 2016.

26. Shuai Wang, Pei Wang, and Dinghao Wu. Composite software diversification. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME '17)*, 2017.
27. Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu. Turing obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM'17)*, 2017.
28. Zhi Wang, Chunfu Jia, Min Liu, and Xiaoxu Yu. Branch obfuscation using code mobility and signal. In *Proceedings of 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW'12)*, pages 553–558, 2012.
29. Martin R. Woodward, Michael A. Hennell, and David Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, 5(1):45–50, January 1979.
30. Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *Proceedings of the 19th Information Security Conference (ISC'16)*, pages 323–342, 2016.