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

OBFUSCATION WITH TURING MACHINE

A Thesis in
Information Sciences and Technology
by
Yan Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2017

The thesis of Yan Wang 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

Obfuscation is an important technique to protect software from adversary analysis. Control flow obfuscation effectively prevents attackers from understanding the program structure, hence impeding a broad set of reverse engineering activities. In this thesis, a novel control flow obfuscation method is proposed. It employs Turing machines to simulate the computation of branch conditions. By weaving the original program with Turing machine components, program control flow graph and call graph would become more complex. Moreover, due to the computation complexity of a Turing machine, program execution flow would become much more complicated and resilient to advanced reverse engineering approaches through symbolic execution and concolic testing.

A prototype tool based on the proposed technique is implemented. Comparing with previous work, the proposed control flow obfuscation technique bears three distinct advantages. 1). Complexity: the complicated implementation of a Turing machine makes it hard for attackers to understand the program control flow structure. 2). Universality: theoretically, Turing machines can encode any computation. The obfuscation is built on top of the LLVM intermediate representation so the application scope is broadened to almost every language with an LLVM front-end compiler. 3). Resiliency: our obfuscation is shown to be very resilient to advanced analysis tools. We have evaluated the method in terms of functionality correctness, potency, resilience, stealth, and cost, respectively. The experimental results show that the proposed technique can obfuscate programs in stealth with good performance and robustness.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor, Dr. Dinghao Wu for his help through my two years Master study. He not only taught me how to do research projects but also generously gave me valuable advice on my long term career development. It is such an honor to be one of his students.

I would also like to thank Dr. Danfeng Zhang and Dr. Zihan Zhou for being my committee members. They gave me insightful comments and suggestions which made my thesis better.

I would like to express my gratitude to my labmates, especially Shuai Wang and Pei Wang. They helped me with their experience and expertise to overcome obstacles in my thesis research. Besides, I want to thank all of them for two years company in State College, it was a pleasure working with them.

I would like to say thank you to my family for their encouragement and support. Without them, I would never finish my Master study successfully.

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

Obfuscation is an important technique for software protection. Attackers could take advantage of the state-of-the-art techniques [21–23] to recover program source or high-level code from executables, exploit software vulnerabilities, and steal algorithm implementations. Software obfuscation is mostly designed to impede such (malicious) reverse engineering process. It is also used by malware developers to hide their malicious intent.

Recently, software security has drawn more and more attention because of, for example, infamous ransomware attacks and severe vulnerabilities such as the “WannaCry” incidence [34] and the OpenSSL “Heartbleed” bug [33]. All of these malware programs exploit vulnerabilities inside a program. To launch such attacks, attackers usually need to recover the control flow structures of the victim programs first. Symbolic execution and concolic testing are widely-adopted techniques to cover execution paths and explore program structure [1, 30–32]. Typical symbolic execution engines such as SAGE [17] and KLEE [19] could yield new input which leads to a new execution path by solving branch conditions with a constraint solver. After all execution paths are traversed, control flow graph of the program could be restored with the traversing information. Such tools have been proven to be very effective in analyzing program control flow structures [25].

Hence, a lot of anti-reverse engineering research has focused on preventing adversaries from analyzing important path conditions in a program [5, 7–9, 20]. Control flow obfuscation is one of these cutting-edge techniques to combat these reverse engineering tools. Control flow obfuscation aims at hiding path conditions and complicating the execution flow within a program. By rewriting or adding extra control flow components, the program path conditions become difficult or even impossible to analyze. Existing research [2] have demonstrated the effectiveness of control flow obfuscation.

In this thesis, we propose a novel control flow obfuscation method which leverages Turing machine to compute path conditions. The *Church-Turing thesis* [24] states that the power of Turing machines and λ-calculus is the same as algorithms, or the informal notion of effectively calculable functions. Formally, Turing computable, λ-computable, and general recursive functions are shown to be equivalent, and informally, the thesis states that they all capture the power of algorithms or effectively calculable functions. This means any functional component of software can be re-implemented as or transformed into a Turing machine; the replaced code component and its corresponding semantic equivalent Turing machine is called *Turing Equivalent*.

Our method is to simulate important branch condition statements in a program with semantically equivalent Turing machines. A Turing machine behaves as a state machine so it would bring in a large amount of extra control flow transfers and basic blocks to the overall program control flow graph. Moreover, a typical Turing machine leverages a transition table to guide the computation, and such transition table-based execution would introduce additional computations and make the overall execution flow much more complicated. We envision the proposed technique would largely complicate the protected program, and also bring in new challenges for reverse engineering tasks. Our method can also be used to obfuscate other computation as well.

To obfuscate a program through the proposed Turing machine obfuscation technique, we first translate the original program source code into a compiler intermediate representation. Our Turing machine obfuscator then selects branch condition statements for transformation; the transformed instructions will invoke its corresponding Turing machine component, which is semantically equivalent to the original branch condition. After finishing the execution in the Turing machine "black box", the execution flow returns back to the original instruction, with a return value which determines the branch selection. Inspired by previous work [4], we evaluate our obfuscator regarding five aspects which are functionality correctness, potency, resilience, cost, and stealth. Results indicate that Turing machine obfuscator could effectively obfuscate commonly-used software with acceptable cost and robustness.

This thesis is organized as follows. Section 2 discusses related works on obfuscation, especially control flow obfuscation. Section 3 presents the overall design of Turing machine obfuscator. Obfuscator implementation is discussed in Section 4. Section 5 presents the evaluation result of our proposed technique. We further present discussion in Section 6, and conclude the theis in Section 7.

In Turing machine obfuscator, we collect all comparison instructions for branch predicate instructions (e.g., `icmp`). Through use-def chain analysis, we also locate all instructions that determine the value of the conditional branch instruction, all these instructions are deemed as transformation candidates.

Chapter 2
Related Work

Generally speaking, reverse engineering techniques are categorized into static track and dynamic track. To battle static reverse engineering, researchers usually focus on hardening disassembling and decompiling process. To combat the dynamic reverse engineering techniques such as concolic testing, sensitive conditional transfer logic must be hidden from adversaries. Control flow obfuscation has been proved effective in this scenario.

Sharif et al. [5] propose a technique to rewrite certain branch conditions and encrypt code components that are guarded by such conditions. Branch conditions that are dependent on the input are selected and branch condition outputs are transformed with a hash function. Moreover, the code component which is dependent on a transformed condition would be encrypted; the encryption key is derived from the input which satisfies the branch condition. In general, their technique focus on selectively translate branch conditions that are dependent on the input, which could leave many branch conditions unprotected. Also, since the branch condition statement itself is mostly untouched (only the boolean output is hashed), the original branch condition code is still in the obfuscated program, which could be leveraged to reveal the original semantics.

Popov et al. [7] propose to replace unconditional control transfer instructions such as `jmp` and `call` with “signals”. Their work is used to impede binary disassembly, usually the starting point of reverse engineering. Moreover, dummy control transfers and junk instructions are also inserted after the replaced control transfers. This method is effective in fooling disassemblers in analyzing unconditional transfers but it becomes mal-functional where the conditional transfers need to be protected as well. Another related work proposes to protect control flow branches leveraging a remote trusted third party environment [8]. In general, their technique mostly introduces notable network overhead and also relies on trusted network accessibility which may not be feasible in practice.

Ma et al. [2, 28] propose to use the neural network to replace certain branch condition statements; the proposed technique is evaluated to conceal conditional instructions and impede typical reverse engineering analysis such as concolic testing. Although the idea is promising and the experimental results indicate the effectiveness to a certain degree, in general the neural network approach may not be suitable for security applications. To the best of our knowledge, neural network works like a black box; it lacks a rigorous theoretical foundation to show a correct result can always to generated given an input. In other words, neural networks may yield results which lead to an incorrect branch selection. Overall, neural networks introduce complexity as well as (unwanted) uncertainty to the transformed programs. In addition, we notice that neural network usually consumes too much memory in transforming non-trivial programs.

Chapter 3

Turing Machine Obfuscator

3.1 Design Overview

In a program, a branch condition statement compares two operands and selects a branch for control transfer based on the comparison result. As aforementioned, Turing machine has been proved to be able to simulate the semantics of any functional component of a program. Hence, any program branch condition statement can be modeled by a Turing machine. Taking advantage of its powerful computation ability as well as execution complexity, we propose to employ Turing machine to obfuscate branch condition statements (the branch condition statement is referred as "branch predicate" later in this thesis since its output is usually a boolean value) in a program. A Turing machine obfuscated branch condition statement is shown in Figure 3.1. Instead of directly computing a boolean value through a comparison instruction, we feed a Turing machine with the inputs (the value of operands) and let the Turing machine to simulate the comparison semantics.

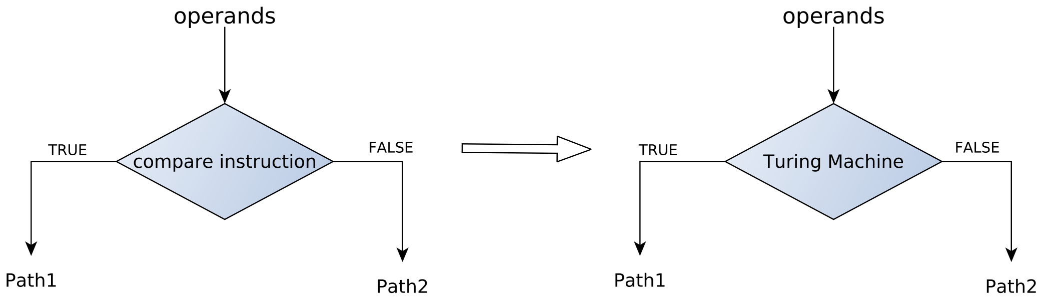


Figure 3.1: Obfuscate a branch condition statement through a Turing machine.

3.2 Turing Machine

As shown in Figure 3.2, a typical Turing machine consists of four components:

- An infinite-long tape which contains a sequence of cells. Each cell holds a symbol defined in the tape alphabet (the alphabet is introduced shortly). In this work, our proposed Turing machine obfuscator would dynamically allocate new tape cells to construct an infinite tape to store intermediate results.

- A tape head which could perform `read`, `write`, `move left` and `move right` operations over the tape.

- A state register used to record the state of the Turing machine. Turing machine states are finite and defined in the transition table.

- A transition table that consists of all the transition rules defining how a Turing machine transfers from one state to another.

Although simple, Turing machine model resembles a modern computer in several ways. The head is I/O device. The infinite tape acts as the computer memory. The transition table defines the mission of this Turing machine which is like the

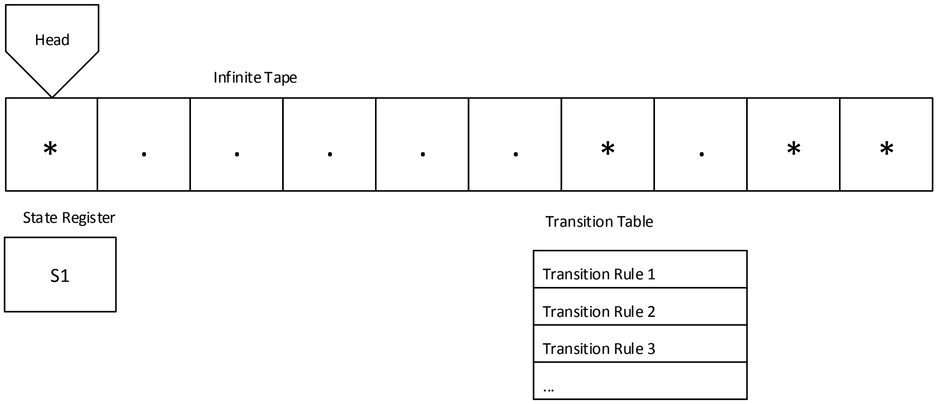


Figure 3.2: Turing machine components.

program code and data. Hence, Turing machine is also deemed as the foundation of modern computer science development.

3.2.1 Transition Table

A transition rule could be represented by a five-element tuple (S_c, T_c, S_n, T_n, D) where:

- S_c is the current Turing machine state
- T_c is the current tape cell symbol read by the head
- S_n is the new Turing machine state
- T_n is the symbol head writes to the current tape cell
- D is the direction the head should move (i.e., “left” or “right”)

In general, every five-element tuple represents a transition table rule shown in Figure 3.2.

3.2.2 Turing Machine Encoding

Initially, Turing machine is at the "start" (S_0) state and the tape records the Turing machine input. Consistent with existing Turing machine simulator project [6], blank symbol is denoted as "*" on the tape, while the length of "·" is used to encode an operand of integer type (our current implementation only focuses on operand of integer type, we present further discussion on this in §4.3). For instance, integer 5 is represented as five continuous "·" on the tape. Note that a Turing machine could be encoded with various of ways, our prototype represents only one of them. Turing machine with different encoding strategies operates with totally distinct execution pattern. This also makes Turing machine obfuscation hard to be analyzed.

In general, our Turing machine tape alphabet includes two symbols, i.e., $\{\cdot, *\}$. The tape in Figure 3.2 shows an initial state of a Turing machine. The head of Turing machine is placed on the leftmost cell. Different integer operands are separated by a blank symbol "*". Operands encoded on the tape of Figure 3.2 are five and one. When Turing machine starts to run, the head reads the current tape cell, combines with current state register to locate a transition rule in the transition table, and then moves to next state accordingly.

3.2.3 Turing Machine Execution

The Turing machine keeps running step by step directed by the transition table until it reaches a `Halt` state. On the other hand, Turing machine may keep running forever since the process of solving some problems cannot terminate. In our research, we implement a Turing machine to simulate branch predicates (i.e., simple algebra computations) so it should always reach a `Halt` state. When reaching the `Halt` state, the machine stops running and the computation result is shown on the tape.

Current State	**Current Symbol**	**New State**	**New Symbol**	**Direction**
S_0	*	S_0	*	Right
S_0	.	S_1	.	Right
S_1	*	S_2	.	Right
S_1	.	S_1	.	Right
S_2	*	S_3	*	Left
S_2	.	S_2	.	Right
S_3	*	S_3	*	Left
S_3	.	S_4	*	Left
S_4	*	$Halt$	*	-
S_4	.	S_4	.	Left

Table 3.1: Transition table of the `add` operation in a Turing machine.

Table 3.1 shows a transition table example, which supports a Turing machine to conduct the addition operation in our implementation.

3.2.4 Addition Turing Machine

In this section, we elaborate the design of the addition Turing machine which simulates the semantics of the `add` operation. Other Turing machines (e.g., subtraction and multiplication Turing machine) used in this research are designed in a similar way. Through constructing this machine, we essentially build rules which could concatenate two series of · cells on the tape together. Take initial tape in Figure 3.2 as an example. Following the rules in table 3.1, after a sequence of read and write operations based on the transition table, left operand (integer value 5) and right operand (integer value 1) which are separated by a blank symbol "*" are merged into a long series of · cells on tape; the length of the outcome dot cells is 6, which represents integer value 6 as shown in Figure 3.3.

Reading transition table directly is difficult for a human being. To represent an understandable description on how the transition table for the addition operation works, we summarize the transition table logic and represent it in an algorithm

*	·	·	·	·	·	·	*	*

Figure 3.3: Turing machine execution result.

description. Algorithm 1 describes the transition table; in fact it states a method to combine two sequences of dot cells on tape into a longer sequence of cells. Following the algorithm, the isolator cell (i.e., the blank symbol) is written to · when the Turing machine finally enters the "Halt" state.

Algorithm 1 Description of the addition transition table.

```
procedure
    head ← the blank cell before the left operand starting cell
    while head != the blank cell after the right operand do move right
    move left
    the last dot cell of the right operand ← blank symbol
    while head != the blank cell within these two operands do move left
    the blank cell ← dot
    while head != the blank cell before the left operand do move left
    Halt;
```

3.2.5 Turing Machine of Other Operations

Since our Turing machine obfuscator essentially focuses on obfuscating branch predicates which usually involve with arithmetic operations, Turing machine obfuscator also needs to provide other transition tables of arithmetic operations (such as $-, \times, \div$).

To implement the arithmetic operations, besides the addition transition table shown in Table 3.1, we construct three more transition tables for subtraction, multiplication and division operations. Their transition tables are relatively more complex than Table 3.1. Actually in our implementation, we build transition table consisting of 16, 34 and 80 transition rules entries for subtraction, multiplication and division Turing machines, respectively. As for the comparison operations such as $\leq, \geq, \neq$, we take advantage of the subtraction Turing machine to calculate them. Based on the calculation result, a boolean value is returned. In sum, we construct 4 transition tables, with overall 140 transition table entries.

3.3 Universal Turing Machine

While a Turing machine could perform powerful algorithm simulation, its computation ability is predetermined by its initial tape state and intrinsic transition table. For instance, a Turing machine capable of doing addition operation could only simulate the "add" operation since other operations would have very different transition rules. That is, an "add" Turing machine could not represent the "subtract" operations. Also, since the initial state needs to be encoded on the tape before the computation, a Turing machine encoded with $2 + 3$ could not conduct addition operation for $5 + 6$.

In non-trivial programs, branch predicate could include various arithmetic and comparison operations, and many of these expressions would correspond to different Turing machines. Hence, we need a unified translator to represent arbitrary computations. Universal Turing machine is designed to simulate arbitrary computations. As shown in Figure 3.4, both input data and transition table are initialized on tape as a single tape universal Turing machine. As a result, all the information needed for computations exists. In some sense, a universal Turing machine acts as the interface for us to employ Turing machines of different semantics.

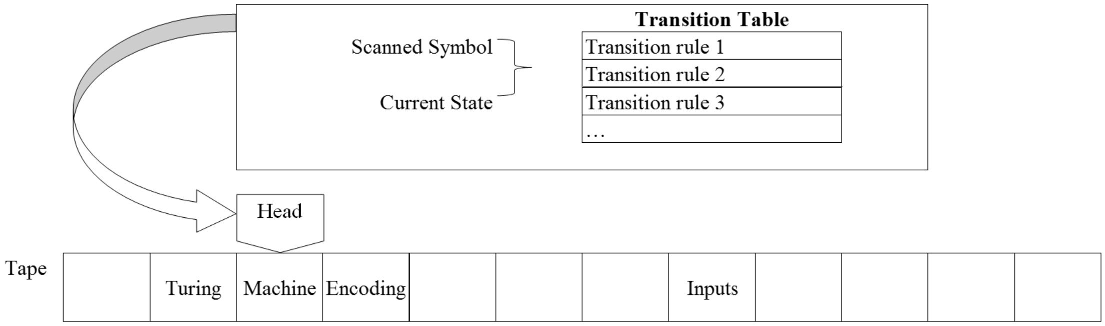


Figure 3.4: Universal Turing machine.

Universal Turing machine bears the essence of the modern computer which is being programmable. Through storing different transition tables and inputs on the tape, a universal Turing machine can actually perform the semantic equivalent computation to arbitrary programs; as aforementioned, such Universal Turing Machine and the replaced expression are *Turing Equivalent.* In our Turing machine obfuscator, different branch predicates invoke a unified interface, which bridges the obfuscated instruction and a universal Turing machine.

Chapter 4 Implementation

Our proposed obfuscator is made up of several components including a universal Turing machine model implemented in C and several transformation passes based on the LLVM compiler suite [18]; As shown in Figure 4.1, our Turing obfuscator performs a three-phase process to generate the obfuscated output. The first step translates both target program and universal Turing machine source code into LLVM intermediate representation (IR). The obfuscator then iterates IR instructions to identify obfuscation candidates (the second phase). Given all the transformation candidates, we then perform the obfuscation transformation (the third phase). The instrumented IR codes are further compiled into the final executable output. We implement the universal Turing machine model with in total 580 lines of C code and LLVM passes with 341 lines of C++ code. We now elaborate on each phase in details.

4.1 Phase One: Translate Source Code to IR

As aforementioned, we first compile the target source program into LLVM IR with an appropriate LLVM front end compiler; the obfuscation transformation is

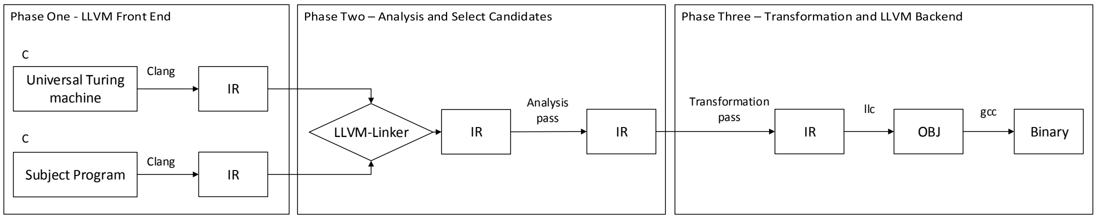


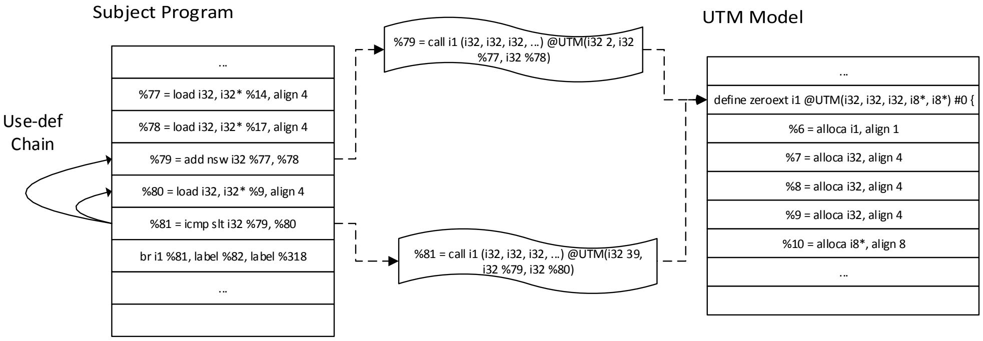
Figure 4.1: Workflow of the Turing machine obfuscator.

performed on the IR level. Considering a broad set of front end compilers provided by LLVM which can turn programs written by various programming languages into its IR, this IR-based implementation could broaden the application scope comparing with previous work [2, 8, 28]. Since we employ C programs for the evaluation, Clang (version 5.0) is used as the front end compiler in this thesis.

4.2 Phase Two: Collect Transformation Candidate

The LLVM Pass framework is a core module of the LLVM compiler suite to conduct analysis, transformation and optimization during the compile time [18]. We build a pass within this framework to iterate and analyze every IR instruction in each module of the input program. During the analysis pass, our Turing machine obfuscator locates all transformation candidates on the IR instruction level.

Locate Candidate Predicates While the proposed technique is fundamentally capable of obfuscating any program component, the implementation currently focuses on branch predicate since control-flow obfuscation is efficient to defeat many reverse engineering activities (§1). In general, the transformation candidate set includes 10 kinds of branch predicate instructions as: equal, not equal, unsigned less than, unsigned greater than, unsigned less or equal, unsigned greater or equal,



```
...
%77 = load i32, i32* %14, align 4
%78 = load i32, i32* %17, align 4
%79 = add nsw i32 %77, %78
%80 = load i32, i32* %9, align 4
%81 = icmp slt i32 %79, %80
br i1 %81, label %82, label %318
...
```

```
%79 = call i1 (i32, i32, i32, ...) @UTM(i32 2, i32
%77, i32 %78)
```

```
%81 = call i1 (i32, i32, i32, ...) @UTM(i32 39,
i32 %79, i32 %80)
```

```
...
define zeroext i1 @UTM(i32, i32, i32, i8*, i8*) #0 {
%6 = alloca i1, align 1
%7 = alloca i32, align 4
%8 = alloca i32, align 4
%9 = alloca i32, align 4
%10 = alloca i8*, align 8
...
```

Figure 4.2: Obfuscation transformation for an `icmp` instruction. "UTM" stands for universal Turing machine.

signed less than, signed greater than, signed less or equal, signed greater or equal.

4.3 Phase Three: Obfuscation Transformation

The second phase provides all the eligible transformation candidates. We further build another transformation pass within the LLVM Pass framework to perform the obfuscation transformation. As shown in Figure 4.2, predicate instructions are obfuscated; we rewrite the instructions into function calls to the universal Turing machine interface. The computation of the branch predicate is launched inside the Turing machine, and the computation result is passed to a register which directs the associated path selection. Our technique is able to obfuscate all the branch predicates in a program or only transform a subset of (security sensitive) candidates. Such partial obfuscation is denoted as "obfuscation level" and we present discussion on this shortly.

For an obfuscated predicate, our current "transform to function call" implementation utilizes the boolean return value to select a branch for control transfer. On the other hand, we notice existing work (e.g., [2, 28]) leverages a cross-procedure jump at this step; an indirect jump from the black box of obfuscation component to a selected branch. We present a further discussion on both control transfer strategies in §6.

Operand Type In general, a branch predicate instruction can have either pointer type or numerical data type (i.g., integer or float type). While the proposed technique is generally capable of translating branch predicate with any operand type, considering processing operands of pointer (and float) type would bring in additional complexity, our current prototype is designed to only handle operands of integer type. Actually our tentative study shows that most of the branch predicate instructions would have operands of integer type, hence, this implementation choice is indeed able to handle most of the real-world cases. On the other hand, we emphasize there is no additional research challenge to extend our technique to handle other cases. We leave it as one future work to provide such functionalities.

Def-use Chain Analysis Since our analysis is performed on IR expressions of three-address form, one branch predicate in the original program shall be translated into a sequence of IR instructions. Hence, to perform a faithful obfuscation of one branch predicate, we need to first identify a "region" of IR instructions that are translated from this predicate.

As shown in Figure 4.2, we perform def-use analysis to recover such "region" information. In particular, given a comparison IR instruction (which indicates one branch predicate and the end of the "region"), we calculate the use-def chains of its two operands, respectively. The identified instructions which provide the "definition" information of these two operands will be included in the "region". After the def-use analysis, we obfuscate all the instructions in the "region".

Obfuscation Level Obfuscation level is an indicator which weighs how much of a program is transformed by the obfuscation pass. Consistent with previous work [3], the obfuscation level is defined as the ratio between the obfuscated instruction and total eligible candidates:

$$O = M/N$$

M is the number of instructions transformed by the obfuscation pass. N is the number of all the transformable instructions (i.e., the branch predicate instructions identified in §4.2).

Chapter 5

Evaluation

Inspired by previous research [2, 4, 28], we evaluate our Turing machine obfuscator based on five metrics which are *functionality*, *potency*, *resilience*, *stealth* and *cost* respectively. Potency weighs the complexity of the obfuscated programs, which is straightforward to show how competent an obfuscator is. A good obfuscator also needs to protect itself from being deobfuscated; to measure how well an obfuscated program is resilient to automatic deobfuscation techniques, we evaluate the resilience of our Turing machine obfuscator. Moreover, in the battle against experienced attackers, obfuscated programs should not be too distinguishable from its origins otherwise it would be easy to be recognized. Hence, we measure the stealth to show how well an obfuscated program resembles the original one. Cost is naturally employed to measure the execution overhead of a software program. While obfuscation would inevitably introduce some performance penalty, we measure the execution time of the obfuscated code to show the overall cost is acceptable.

Two widely-used open source programs are employed in our evaluation: compress tool BZIP2 (version 1.0.6) [10] and regular expression engine REGEXP (version 1.3) [11]. Obfuscation level is an index which represents the ratio between obfuscated instructions and all predicate candidates. In our experiments, the ratio is set as

50% which means half of all conditional transfer candidates are randomly selected and obfuscated.

5.1 Functionality

Both programs evaluated in our research (BZIP2 [10] and REGEXP [11]) provide test cases to verify the functionality of the compilation outputs. In particular, the BZIP2 test cases deliver 3 compression samples and 3 decompression samples, while the REGEXP contains 149 test cases of various regular expression patterns. We leverage those shipped test cases to verify the functionality correctness of our obfuscated programs. For all the evaluated obfuscation levels (i.e., 30%, 50%, 80% and 100%), we report all the obfuscated programs can pass all the test cases, hence preserving the original semantics after obfuscation.

5.2 Potency

Control flow graph (CFG) and call graph provide insights on the general structure of a program and they are the foundation for most static software analysis. With the help of IDA Pro [13], a well-known commercial binary analysis tool, we recover CFG and call graph information from both original and obfuscated binaries. By traversing the graphs, we further calculate the number of basic blocks, number of call graph and control graph edges. We use such information to measure the complexity of a program, which is aligned with previous research [12]. Analysis result are shown in Table 5.1. Comparing the original and obfuscated programs, it can be observed that program complexity is increased in terms of each metric.

We further quantify the Turing machine obfuscated programs w.r.t. the cy-

Table 5.1: Potency evaluation in terms of program structure-level information.

Program	**# of CFG Edges**	**# of Basic Blocks**	**# of Function**
BZIP2	3942	2647	78
obfuscated BZIP2	4195	2828	134
REGEXP	906	619	25
obfuscated REGEXP	1122	773	43

Table 5.2: Potency evaluation in terms of knot and cyclomatic number.

Program	**# of Cyclomatic**	**# of Knot**
BZIP2	1297	5596
obfuscated BZIP2	1369	5720
REGEXP	289	478
obfuscated REGEXP	351	1068

clomatic number and knot number (these two metrics are introduced in [14, 15]). Cyclomatic metric is defined as

$$\text{Cyclomatic} = E - N + 2$$

where E and N represent the number of edges and the number of nodes in a CFG, respectively. Knot number shows the number of edge crossings in a CFG. These two metrics intuitively weigh how complicated a program is in terms of logic diversion number. Results in Table 5.2 shows that both knot number and cyclomatic number notably increase after Turing machine obfuscation. Overall, we interpret Table 5.1 and Table 5.2 as promising results to show program becomes more complex after the Turing machine obfuscation.

Besides picking 50% as the obfuscation level in evaluating potency, we also conduct experiments with obfuscation levels as 30%, 80% and 100%. Figure 5.1 shows the number of call graph edges regarding different obfuscation levels. Observation shows that with a higher obfuscation level, the number of call graph edges increases.

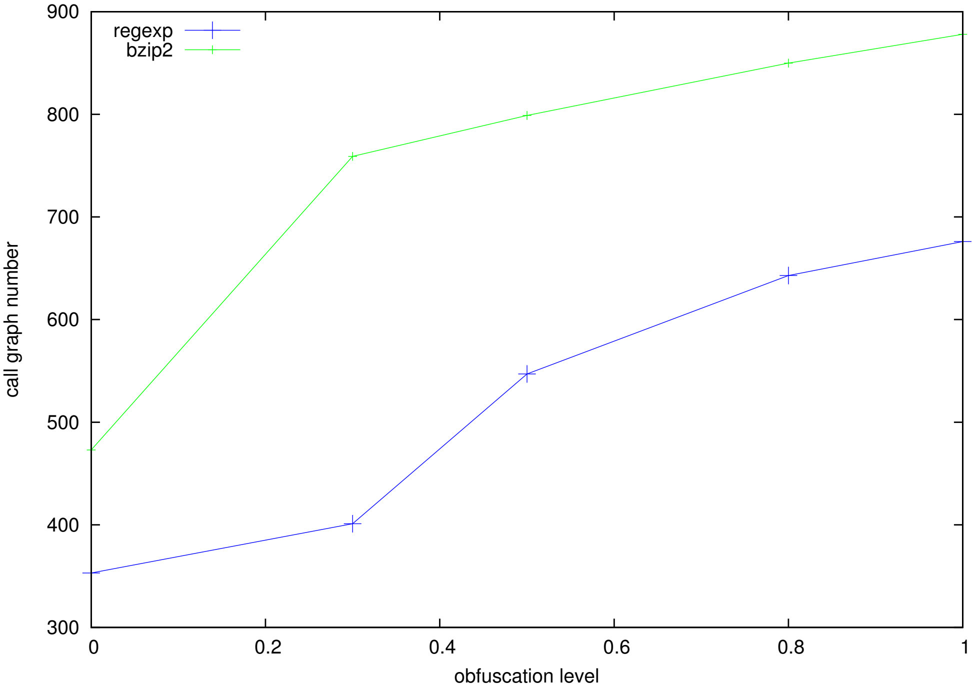


Figure 5.1: Number of call graph edges in terms of different obfuscation levels.

We interpret the results that the obfuscated program become more complicated with the obfuscation level increasing.

5.3 Resilience

A good obfuscation technique should resist deobfuscation tools as well. Concolic testing is an advanced deobfuscation technique aiming at finding bugs or vulnerabilities in software through the mixture of symbolic execution and concrete execution. Whereas, it is also used by adversaries to analyze or restore software control flow graph [1, 25, 26]. KLEE [19] is a symbolic execution tool based on the LLVM platform. Starting with a concrete input it could explore a certain execution path. By negating the last branch condition, it would diverge from the last "crossing" node and explore a new path. KLEE repeats this process unceasingly like doing

```
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}

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

Figure 5.2: KLEE sample code used in our evaluation. All the path conditions are obfuscated.

depth first search until all execution paths are found. With the help of KLEE, it would be easy to conduct automated deobfuscation. We choose KLEE as the deobfuscation tool to evaluate resilience of Turing Machine obfuscator. We used a piece of sample code from KLEE [29] as the subject program (the sample code is shown in Figure 5.2). The subject program need to be converted to IR codes since KLEE works on IR level.

KLEE could detect three paths in the original subject program as expected. Based on different value of x, this program may traverse branches in which x equals 0, x is less than 0 and x is greater than 0, respectively. In contrast, after subject program is obfuscated by our Turing machine obfuscator, we report that KLEE could only figure out **one** path. We interpret the evaluation result that Turing machine obfuscator can impede automated deobfuscation tools from restoring the structure of a program.

Due to limited information released by KLEE, we could not figure out the underlying reason that leads to the failure of KLEE. Since Turing machine obfuscator makes the conditional branches more complicated, we envision that the internal constraint solver employed by KLEE is unable to yield a proper symbolic input

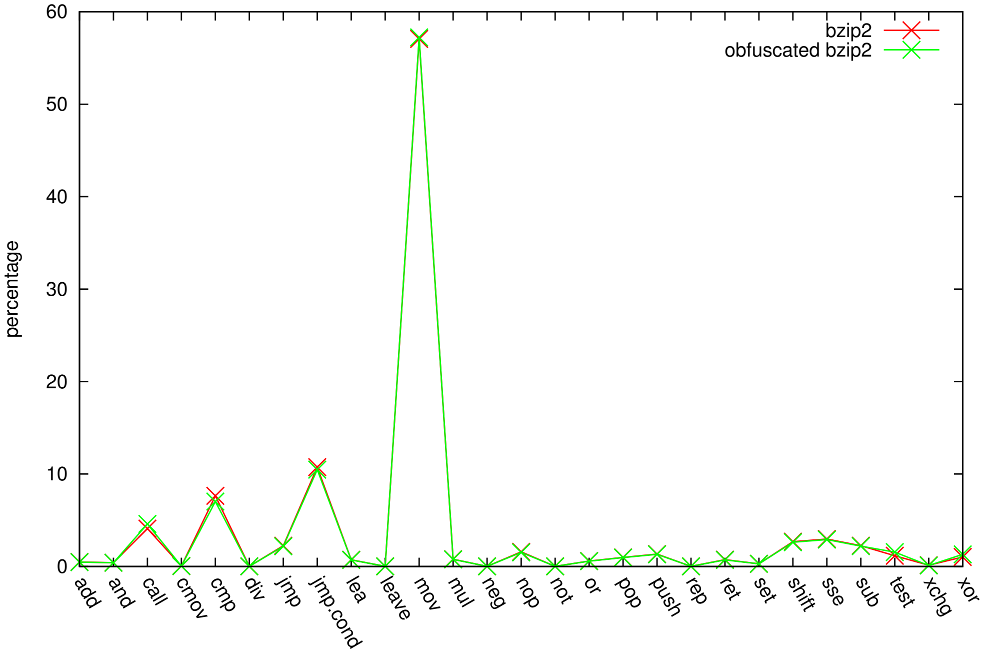


Figure 5.3: BZIP2 instruction distribution comparison.

which "drill" into the branches protected by our Turing machine obfuscator.

5.4 Stealth

As mentioned in the beginning, software obfuscation technique should not only combat automated deobfuscation tools, but also manual deobfuscation methods. In the evaluation of stealth, Wang et al. [3] compare the instruction distributions of the original and obfuscated programs. If instruction distribution of the obfuscated program is distinguishable from its original program (e.g., `call` or `jmp` instruction proportions are abnormally high), it would be an indicator that the program is manipulated. We adopted this metric to evaluate our Turing obfuscator. Obfuscation level for stealth evaluation is set to 50%.

Consistent with previous research [3], we put assembly instructions into 27

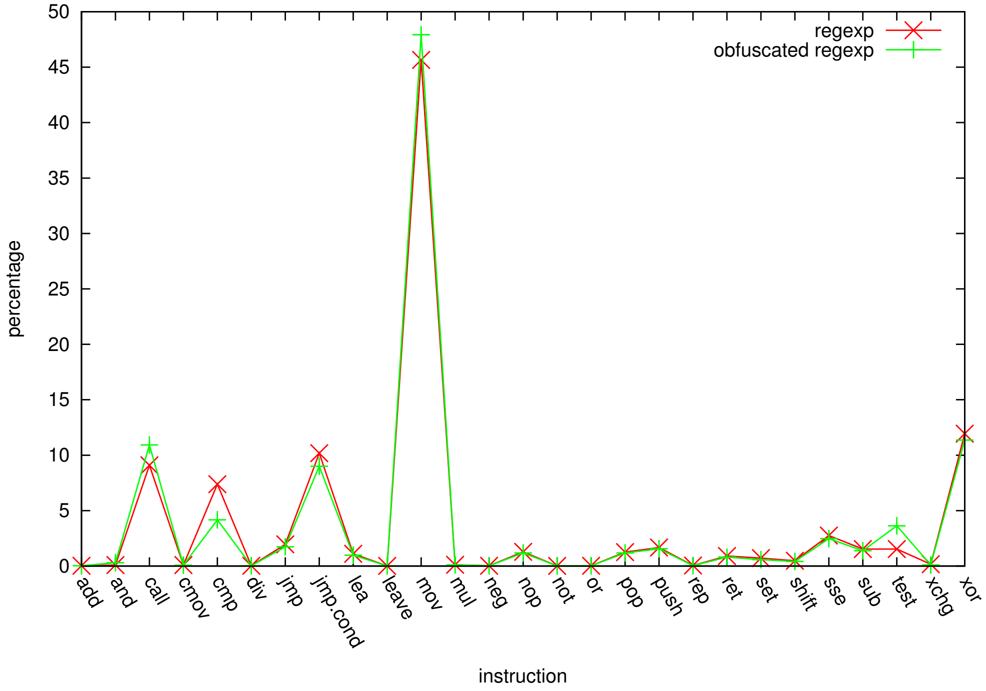


Figure 5.4: REGEXP instruction distribution comparison.

different categories. Figure 5.3 and Figure 5.4 present the instruction distribution of the original and obfuscated programs (BZIP2 and REGEXP). Experiment results indicate that the instruction distribution after obfuscation is very close to the origin distribution. In sum, small instruction distribution variation is a promising result to show the proposed technique would obfuscate programs in a stealthy way.

5.5 Cost

Software running cost is another critical factor in evaluating an obfuscation technique. In most obfuscation research work, execution cost is inevitably increased because obfuscation would bring in extra instructions. Measuring the execution time is a convincing way to evaluate to the cost.

In our evaluation, both original and obfuscated programs are executed on a server

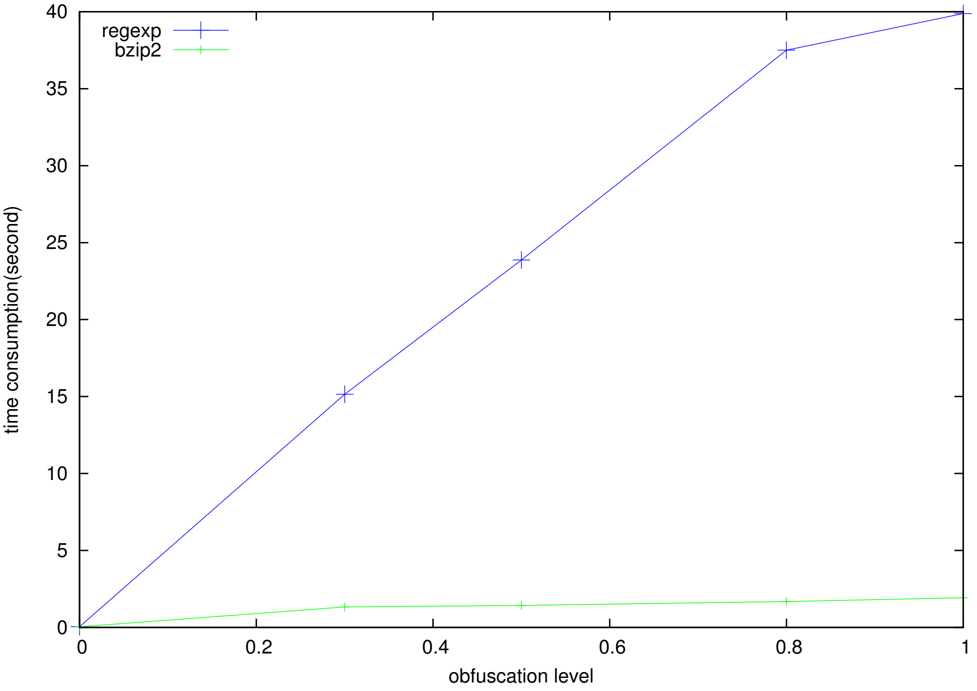


Figure 5.5: Execution overhead in terms of different obfuscation levels.

with 2 Intel(R) Xeon(R) E5-2690 2.90GHz processors and 128GB system memory. BZIP2 is used to compress three different sample files and regular expression engine REGEXP runs 149 samples provided in its shipped test cases. We run each program three times and record the average time cost as the final result.

Figure 5.5 shows that for both test cases, the execution slowly grows w.r.t the increase of obfuscation levels. As expected, program takes more time to execute with more instructions are obfuscated. On the other hand, we interpret the overall time cost is still confined to a reasonable level. We also notice that there exists a difference between slopes of the two curves. Turing obfuscator randomly obfuscates candidate instructions within the program before execution. Hence the transformed instructions may not be indeed executed in the runtime. Difference between two curve slopes is probably due to such uncertainty of execution. In addition, some further study on the source code show that REGEXP employs more recursive calls than BZIP2, thus may lead to more invocations of the Turing machine component and contribute to the performance penalty.

Chapter 6 Discussion

To provide more insights and guideline for further adoption of our proposed technique, we discuss the multiple aspects of the proposed Turing machine obfuscation technique in this section.

6.1 Complexity

In general, Turing machine model is a powerful calculator that is capable of solving any algorithm problem. Note that even a simple operation (e.g., “add”) may lead to the change of Turing machine states for hundreds of times; every “move left” and “move right” operation lead to the tape modification and “read tape” or “write tape” operations.

Considering Turing machine as a state machine, it is hard—if possible at all—for adversaries with manual reverse engineering to follow the calculation logic without understanding the transition table rules and state variables. In addition, automated deobfuscation tools (e.g., KLEE) can also be defeated due to the intrinsic complexity of a Turing machine. As reported in our resilience evaluation (§5.3), the constraint solver of KLEE failed to yield proper inputs to cover two of three execution paths.

6.2 Application Scope

Previous obfuscation work [5] usually targets one or several specific kinds of predicate expressions. Also, most of them performs source code level transformations for specific kind of program languages [3]. Turing obfuscator broadens the application scope to any kind of conditional expression. In addition, it works for programs written in any language as long as they could be transformed into the LLVM IR. Considering a large portion of programming languages have been supported by LLVM, we envision Turning machine obfuscator would serve to harden many softwares implemented with various kinds of programming languages.

6.3 Branch Selection Techniques

As previously presented, our current implementation rewrites path condition instructions to invoke the Turing machine component. While it is mostly impossible for attackers to reason the semanic of the Turing machine code, return value of executing the Turing machine component is observable (since the predicate computation is modeled as a function call to the Turing machine component). Certain amount of information leakage may become feasible at this point.

We notice that existing work [2, 28] proposes a different approach at this step; control flow is directly guided (via `goto`) to the selected branch from the neural network obfuscator. While this approach seems to hide the explicit return value, we argue such technique is not fundamentally more secure since the hidden return value can be inferred by observing the execution flow. Another solution that may be employed to protect the predicate computation result is to use matrix branch logic [27]. Suppose we model a branch predicate with a Turing machine function, the general idea is to further transform Turing machine into a matrix function, and then randomize the matrix branching function. The involved matrix branch logic and randomness shall provide additional security consideration at this step. Overall, we argue the current implementation is reasonable, and we leave it as one future work to present quantitative analysis of the potential information leakage and countermeasures at this step.

6.4 Execution Overhead

During the Turing machine computation, frequent state change would indicate lots of read and write operations. Also, since tape is infinite in Turing machine model, it needs to allocate enough memory to accommodate complex computations. In general, the complexity of Turing machine may be considered as a double edge sword; it impedes adversaries and potentially increases overhead to certain degree. As reported in the cost evaluation (Figure 5.5), we observed non-negligible performance penalty for both cases. One countermeasure here is to perform selective obfuscation; users can mark sensitive program codes and guide the Turing machine obfuscator to only harden those parts. Such strategy would improve the overall execution speed without sacrificing the major security considerations.

Chapter 7
Conclusion

In this thesis, a novel obfuscation technique using a universal Turing machine model is proposed. We have implemented a research prototype, the Turing machine obfuscator, on the LLVM platform and evaluated it on open source software with respect to functionality correctness, potency, resilience, stealth, and cost. The results indicate effectiveness and robustness of the Turing machine obfuscator. We believe the Turing machine obfuscator is a promising and practical obfuscation tool to battle malicious reverse engineering.

Bibliography

[1] Sen, Koushik, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263-272. ACM, 2005.

[2] Ma, Haoyu, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. "Control flow obfuscation using neural network to fight concolic testing." In *International Conference on Security and Privacy in Communication Systems*, pp. 287-304. Springer International Publishing, 2014.

[3] Wang, Pei, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. "Translingual obfuscation." In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pp. 128-144. IEEE, 2016.

[4] Collberg, Christian, 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*, pp. 184-196. ACM, 1998.

[5] Sharif, Monirul I., Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. "Impeding Malware Analysis Using Conditional Code Obfuscation." In *The Network and Distributed System Security Symposium.* 2008.

[6] `http://turingmaschine.klickagent.ch/`

[7] Popov, Igor V., Saumya K. Debray, and Gregory R. Andrews. "Binary Obfuscation Using Signals." In *USENIX Security Symposium*, pp. 275-290. 2007.

[8] Wang, Zhi, Chunfu Jia, Min Liu, and Xiaoxu Yu. "Branch obfuscation using code mobility and signal." In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pp. 553-558. IEEE, 2012.

[9] Wang, Zhi, Jiang Ming, Chunfu Jia, and Debin Gao. "Linear obfuscation to combat symbolic execution." In *European Symposium on Research in Computer Security*, pp. 210-226. Springer Berlin Heidelberg, 2011.

[10] `http://www.bzip.org/`

[11] `https://github.com/cesanta/slre`

[12] Chen, Haibo, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-chung Yew. "Control flow obfuscation with information flow tracking." In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 391-400. ACM, 2009.

[13] `https://www.hex-rays.com/products/ida/`

[14] McCabe, Thomas J. "A complexity measure." *IEEE Transactions on Software Engineering* 4 (1976): 308-320.

[15] Woodward, Martin R., Michael A. Hennell, and David Hedley. "A measure of control flow complexity in program text." *IEEE Transactions on Software Engineering* 1 (1979): 45-50.

[16] Hennie, Fred C. "One-tape, off-line Turing machine computations." *Information and Control* 8, no. 6 (1965): 553-578.

[17] Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In *The Network and Distributed System Security Symposium*, vol. 8, pp. 151-166. 2008.

[18] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, p. 75. IEEE Computer Society, 2004.

[19] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, pp. 209-224. 2008.

[20] Xu, Dongpeng, Jiang Ming, and Dinghao Wu. "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method." In *International Conference on Information Security*, pp. 323-342. Springer International Publishing, 2016.

[21] Ming, Jiang, Dongpeng Xu, Li Wang, and Dinghao Wu. "Loop: Logic-oriented opaque predicate detection in obfuscated binary code." In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 757-768. ACM, 2015.

[22] Lee, Gareth, John Morris, Kris Parker, Gary A. Bundell, and Peng Lam. "Using symbolic execution to guide test generation." In *Software Testing, Verification and Reliability* 15, no. 1 (2005): 41-61.

[23] Molnar, David, Xue Cong Li, and David Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs." In *USENIX Security Symposium*, vol. 9, pp. 67-82. 2009.

[24] Copeland, B. Jack. "The Church-Turing thesis." *Stanford Encyclopedia of Philosophy* (2002).

[25] Sen, Koushik, and Gul Agha. "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools." In *International Conference on Computer Aided Verification*, pp. 419-423. Springer Berlin Heidelberg, 2006.

[26] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, pp. 209-224. 2008.

[27] Garg, Sanjam, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. "Candidate indistinguishability obfuscation and functional encryption for all circuits." *SIAM Journal on Computing* 45, no. 3 (2016): 882-929.

[28] Ma, Haoyu, 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, no. 10 (2016): 2322-2337.

[29] `http://klee.github.io/tutorials/testing-function/`

[30] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In *ACM Sigplan Notices*, vol. 40, no. 6, pp. 213-223. ACM, 2005.

[31] Cadar, Cristian, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: automatically generating inputs of death." In *ACM Transactions on Information and System Security (TISSEC)* 12, no. 2 (2008): 10.

[32] King, James C. "Symbolic execution and program testing." In *Communications of the ACM* 19, no. 7 (1976): 385-394.

[33] `http://heartbleed.com/`

[34] `https://www.bu.edu/eng/2017/05/18/protection-from-ransomware-like-wannacry/`