Turing Obfuscation

Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu

College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{ybw5084,szw175,pxw172,dwu}@ist.psu.edu

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 efforts. In this paper, we propose a novel control flow obfuscation method which 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 can become much more complicated. In addition, due to the runtime computation complexity of a Turing machine, program execution flow would be highly obfuscated and become resilient to advanced reverse engineering approaches via symbolic execution and concolic testing.
We have implemented a prototype tool for Turing obfuscation. Comparing with previous work, our control flow obfuscation technique delivers three distinct advantages. 1). Complexity: the complicated structure of a Turing machine makes it difficult for attackers to understand the program control flow. 2). Universality: Turing machines can encode any computation and hence applicable to obfuscate any program component. 3). Resiliency: Turing machine brings in complex execution model, which is shown to withstand automated reverse engineering efforts. Our evaluation obfuscates control flow predicates of two widely-used applications, and the experimental results show that the proposed technique can obfuscate programs in stealth with good performance and robustness.

Key words: Software security, control flow obfuscation, reverse engineering, Turing machine

1 Introduction

Most software exploitation and hijacking attacks start by identifying program vulnerable points (e.g., buffer overflow). To launch attacks directly towards executable files, attackers usually need to first perform reverse engineering activities and recover the control flow structures of the victim programs. Moreover, we also notice that automated software analyzers can leverage advanced symbolic and concolic testing techniques to explore execution paths and hence revealing hidden vulnerabilities in binary code [6, 20, 12]. Typical concolic engines [11, 5] could yield inputs which lead to new execution paths by solving branch conditions as

constraints, and such technique has been proved as very effect in understanding program structures [19].

A lot of software security research has focused on preventing reverse engineering activities on program control structures and execution paths [29, 21, 18, 26, 27]. Control flow obfuscation is one of these cutting-edge techniques to combat both static and dynamic reverse engineering tools. Control flow obfuscation largely changes the program control flow structures, and it has been shown as effective to hide path conditions and complicate the execution flow of a program. By rewriting or adding extra control flow components, the program path conditions become difficult or even impossible to analyze.

In this paper, we propose a novel control flow obfuscation method which leverages Turing machine to compute path conditions. The *Church-Turing thesis* [9] 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 semantic equivalent Turing machines. A Turing machine behaves as a state machine which brings in extra control flow transfers and basic blocks to the overall program control flow graph. Moreover, a typical Turing machine leverages transition tables to guide the computation, and such transition table-based execution would introduce complicated execution model and make the program execution much more challenging to analyze. We envision the proposed technique would largely complicate the protected program, and also bring in new challenges for reverse engineering analyzers. In addition, since Turing machine can represent the semantics of any program computation, our method is fundamentally capable of obfuscating any functional component.

To obfuscate a program through the proposed Turing obfuscator, we first translate the original program source code into a compiler intermediate representation. Our Turing machine obfuscator then selects branch condition statements (i.e., branch predicates) for transformation; the transformed statements will invoke its corresponding Turing machine component, which is semantic equivalent to the original branch conditions. After finishing the execution in the Turing machine "black box", the execution flow returns back to the original program point, with a return value to determine the branch selection. Consistent with existing work [8], we evaluate our obfuscator regarding five aspects, namely functionality correctness, potency, resilience, cost, and stealth. Results show that the proposed Turing obfuscator can effectively obfuscate commonly-used software systems with acceptable cost, and impede reverse engineering activities through an advanced symbolic execution analyzer (i.e., KLEE [5]).

The rest of this paper 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 give discussions in Section 6, and conclude the paper in Section 7.

2 Related Work

In general, reverse engineering techniques can be categorized into static and dynamic approaches. To impede static reverse engineering, researchers essentially focus on hardening disassembling and decompiling process. To combat the dynamic reverse engineering techniques such as concolic testing, sensitive conditional transfer logic is proposed to be hidden from adversaries. Control flow obfuscation has been proved effective in this scenario.

Sharif et al. [21] 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. [18] propose to replace unconditional control transfer instructions such as `jmp` and `call` with "signals". Their work is used to impede binary disassembling, the starting point of most reverse engineering tasks. 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 could become mal-functional when 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 [26]. 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. [16, 15] propose to use neural network to replace certain branch condition statements; the propose technique is evaluated to conceal conditional instructions and impede typical reverse engineering analysis such as concolic testing. While the experimental results indicate the effectiveness to certain degree, in general neural network-based 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. We also notice some recent work proposing to translate program components implemented in imperial language

(C/C++) into languages of other computation paradigms. It is argued that by mixing languages of different execution model and paradigms, the complexity of software systems grows and reverse engineering becomes more difficult. Wang et al. [23] presents a general framework to translate C statements into a logic statements written in Prolog. Lan et al. [13] proposes to obfuscate program control flow predicates with functional programming language statements.

3 Turing Obfuscation

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 paper since its output is usually a boolean value) in a program. A Turing machine obfuscated branch condition statement is shown in Fig. 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.

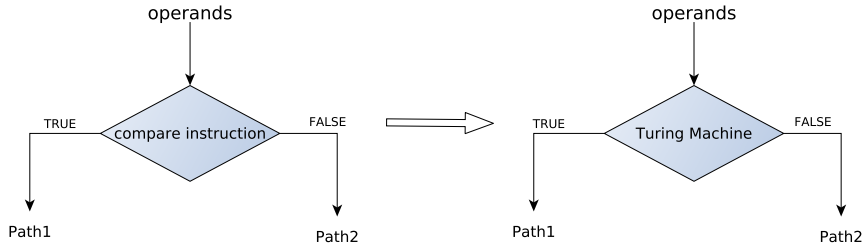

Fig. 1: Obfuscate a branch condition statement through a Turing machine.

3.2 Turing Machine

As shown in Fig. 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, a Turing machine model resembles a modern computer in several ways. The head is I/O device. The infinite tape acts like the memory. The transition table defines the functionality of this Turing machine which is comparable to the application code.

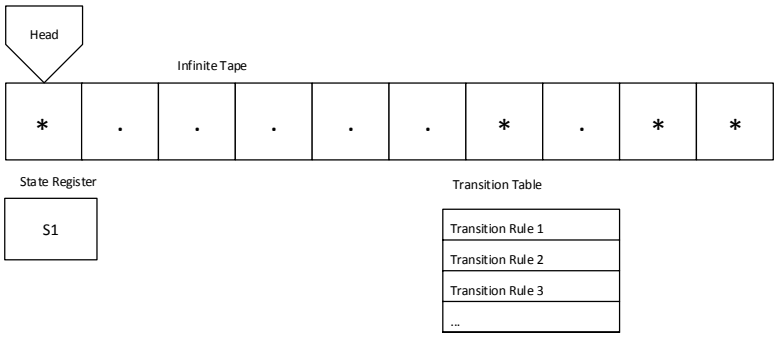

Fig. 2: Turing machine components.

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 towards which the head should move (i.e., "left" or "right").

In general, every five-element tuple represents a transition table rule shown in Fig. 2.

Turing Machine Encoding Initially, Turing machine is at the "start" (S_0) state and tape records the Turing machine input. Consistent with existing Turing machine simulator project [22], blank symbol is denoted as "*" on the tape, while the length of "·" is used to encode an operand. For instance, integer 5 is

represented as five continuous "·" on the tape. Note that a Turing machine could be encoded with various of ways, and our prototype represents only one of them. Turing machine with different encoding strategies operates with totally distinct execution patterns. This also makes Turing machine obfuscation difficult to be analyzed.

In general, our Turing machine tape alphabet includes two symbols, i.e., $\{\cdot, *\}$. The tape in Fig. 2 shows an initial state of a Turing machine. The head of the Turing machine is placed on the leftmost cell. Different operands are separated by a blank symbol "*". Operands encoded on the tape in Fig. 2 are five and one. When Turing machine starts to run, the head reads the current tape cell, combines with the current state register to locate a transition rule in the transition table, and then moves to the next state, accordingly.

Turing Machine Execution The Turing machine keeps running step by step directed by the transition table until it reaches a `Halt` state. Nevertheless, Turing machine may also keep running forever since the process of solving some problems cannot terminate. In our research, we implement a Turing machine to simulate branch predicates 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. Table 1 shows a transition table example, which guides a Turing machine for the addition (i.e., `add`) operation in our implementation.

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

Addition Turing Machine In this section, we elaborate on the design of the addition Turing machine; this machine simulates the semantics of the `add` operation. Other Turing machines (e.g., subtraction and multiplication Turing machines) used in this research are designed in a similar way. Fig. 2 presents a sample initial stage of a tape, and the corresponding addition transition rules are shown in Table 1 (this table will be explained shortly). After a sequence of read and write operations based on the transition table, left operand (integer value 5) and right operand (integer value 1) that are separated by a blank symbol "*"

are merged into a long series of "·" cells on the tape. The length of the output dot cells is 6, which represents the integer value 6 as shown in Fig. 3.

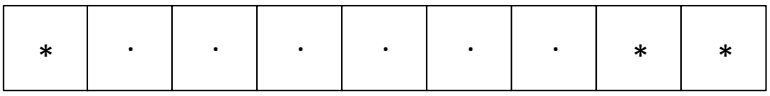

Fig. 3: Execution result of the `add` Turing machine.

Interpreting a transition table could be difficult for a human being. To represent an understandable description on how the addition transition table works, we summarize the transition table rules in an algorithm description. Algorithm 1 describes the transition table of the addition operation; it states a method to combine two sequences of dot cells on the tape into a longer sequence of cells. Following this algorithm, the isolator cell (i.e., the blank symbol) is written to "·" when Turing machine terminates at the "Halt" state.

Algorithm 1 Description of the `add` transition table.

1: **procedure**
2: *head* ← the blank cell before the left operand starting cell
3: **while** head != the blank cell after the right operand **do** move right
4: move left
5: the last dot cell of the right operand ← blank symbol
6: **while** head != the blank cell within these two operands **do** move left
7: the blank cell ← dot
8: **while** head != the blank cell before the left operand **do** move left
9: **Halt;**

Turing Machine of Other Operations Besides the aforementioned addition operation, we also implement transition tables of other arithmetic operations. In particular, we construct three more transition tables for subtraction, multiplication and division operations. Their transition tables are relatively more complex than Table 1. Actually in our implementation, we build transition tables of 16, 34 and 80 transition rules for subtraction, multiplication and division Turing machines, respectively. Comparison operations in a branch predicate (e.g, $\leq, \geq, \neq$) is built on the basis of the subtraction Turing machine, and all the arithmetic operations are used to simulate "dependences" of the comparison operations on the IR level (details are given in §4.3). In sum, we construct 4 transition tables, with overall 140 transition table rules in total.

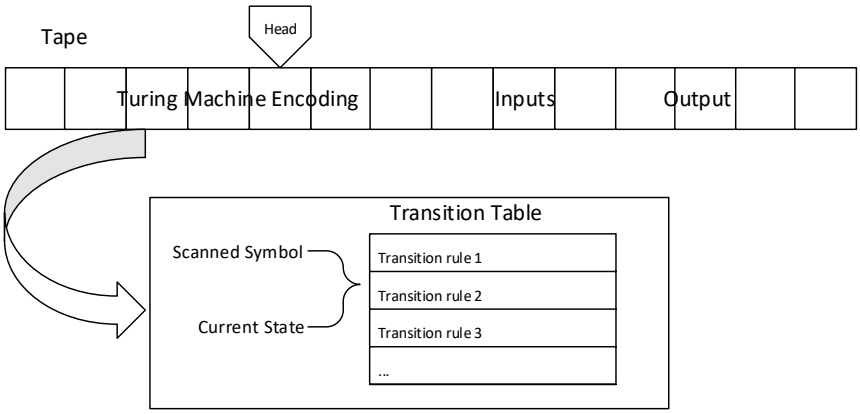

Fig. 4: Universal Turing machine.

3.3 Universal Turing Machine

While a Turing machine could perform powerful algorithm simulation, its computation ability is bounded by its initial tape state and embedded 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 means, 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 represent $5 + 6$.

In non-trivial programs, branch predicate could include various arithmetic and comparison operations, and many of these expressions would lead 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 Fig. 4, the typical design of a Universal Turing machine stores all the transition tables and one table is selected each time according to the semantics of the upcoming computation (e.g., `add`). To maintain the input data, Universal Turing machine dynamically allocates memory cells to initialize one tape before computation. Hence, all the information needed for arbitrary computations exists in the 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 semantic equivalent computation to represent arbitrary programs; as aforementioned, such Universal Turing Machine and the replaced expression are *Turing Equivalent*. In our Turing machine obfuscator design, all the branch predicates invoke a unified interface towards a Universal Turing machine, where a transition table is selected

according to the opcode of the obfuscated instruction, and a tape is constructed to represent the input value.

4 Implementation

Our proposed obfuscator consists several components including a universal Turing machine model and several transformation passes based on the LLVM compiler suite [14]; As shown in Fig. 5, our Turing obfuscator performs a three-phase process to generate the obfuscated output. The first step translates both target program and the universal Turing machine source code into the LLVM intermediate representation (IR). The obfuscator then iterates IR instructions to identify obfuscation candidates (the second phase). After that, we then perform the obfuscation transformation towards all the candidates or a randomly-select portion (the third phase). The instrumented IR code is further compiled into the final obfuscated product. We implement the universal Turing machine model with in total 580 lines of C code and LLVM passes with 341 lines of C++ code.[1] We now elaborate on each phase in details.

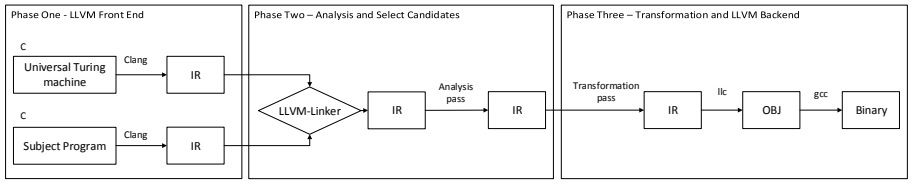

Fig. 5: Workflow of the Turing machine obfuscator.

4.1 Phase One: Translate Source Code to IR

As aforementioned, we first compile the target source program into LLVM IR; the obfuscation transformation is 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 of our tool comparing with previous work [16, 26, 15]. Since we employ C programs for the evaluation, Clang (version 5.0) is used as the front end compiler in this paper.

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 [14].

[1] Please refer to an extended version of this paper for more implementation details [25].

In this step, 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 the 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 predicates since control-flow obfuscation is effective 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, 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. In this step, We build another transformation pass within the LLVM Pass framework to perform the obfuscation transformation. As shown in Fig. 6, predicate instructions are obfuscated; we rewrite instructions into function calls to the universal Turing machine. 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", which will be discussed shortly.

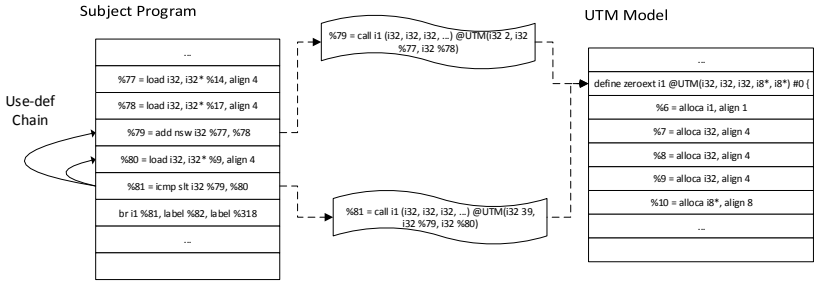

Fig. 6: Obfuscation transformation for an `icmp` instruction. "UTM" standards for universal Turing machine.

For an obfuscated predicate, our current "transform to function call" implementation utilizes a boolean return value to select a branch to transfer. On the other hand, we notice existing work (e.g., [16, 15]) leverages a cross-procedure jump at this step; an indirect jump from the black box of the obfuscation component to a selected branch. We present further discussion on both control transfer

strategies in §6.

Operand Type In general, a branch predicate instruction can have either pointer or numerical data types (i.e., integer or float types). While the proposed technique is generally capable of translating branch predicate of 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, our implementation choice is indeed capable of handling most of the real-world cases. On the other hand, we emphasize extending our technique to handle other cases is only a matter of engineering effort. We leave it as one future work to provide such functionalities.

Def-use Chain Analysis Since our analysis is performed on IR expressions of the 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 is translated from one branch predicate.

As shown in Fig.6, 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 corresponding "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 translate arithmetic instructions in the "region" into function calls to the Turing obfuscator.

Obfuscation Level Obfuscation level is an indicator which weighs how much of a program is transformed by the obfuscation pass. Consistent with previous work [23], the obfuscation level is defined as the ratio between the obfuscated instructions and the total 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).

5 Evaluation

Inspired by previous research [8, 16, 15], we evaluate our Turing machine obfuscator based on four metrics which are *potency*, *resilience*, *stealth* and *cost*, respectively. Potency weighs the complexity of the obfuscated programs, while resilience measures how well an obfuscated program can withstand automated deobfuscation techniques. Stealth is evaluated to show whether the obfuscated

programs are distinguishable regarding its origins, and cost is naturally employed to measure the execution overhead of the obfuscation products. In addition, we also evaluate the functionality correctness of the obfuscated binaries.

Two widely-used open source programs are employed in our evaluation: compression tool BZIP2 (version 1.0.6) [1] and regular expression engine REGEXP (version 1.3) [4]. As aforementioned, obfuscation level is an index which stands for the ratio of obfuscated instructions regarding all the candidates. In our experiments, the ratio is set as 50% unless noted otherwise which means half candidates are randomly selected and obfuscated.

5.1 Functionality Correctness

Both programs evaluated in our research (BZIP2 [1] and REGEXP [4]) are shipped with 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 test cases contain 149 samples 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 represent the general structure of a program and they are the foundation for most static software analysis. With the help of IDA Pro [2], a well-known commercial binary analysis tool, we recover CFG and call graph information from both original and obfuscated binaries. By traversing those graphs, we calculate the number of basic blocks, number of call graph and control graph edges. We use these information to measure the complexity of a (obfuscated) program, which is aligned with previous research [7]. Analysis results are shown in Table 2. Comparing the original and obfuscated programs, it can be observed that program complexity is increased in terms of each metric.

Table 2: 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

We further quantify the Turing machine obfuscated programs in terms of the cyclomatic number and knot number (these two metrics are introduced in [17, 28]). Cyclomatic metric is defined as

$$Cyclomatic = E - N + 2$$

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 measure how complicated a program is in terms of logic diversion number. Results in Table 3 shows that knot and cyclomatic number notably increase for both cases after Turing machine obfuscation. Overall, we interpret Table 2 and Table 3 as promising results to show programs become much more complicated after obfuscation.

Table 3: Potency evaluation in terms of knot and cyclomatic numbers.

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

Besides picking 50% as the obfuscation level in this evaluation, we also conduct experiments with obfuscation levels as 30%, 80% and 100%. Fig. 7 presents the number of call graph edges with the increase of obfuscation levels. Observation shows that with a higher obfuscation level, the number of call graph edges increases. Naturally, obfuscated programs can become more complicated with the growing of obfuscation levels.

5.3 Resilience

In addition to complicate program structures, a good obfuscation technique should be designed to impede automated deobfuscation tools as well. As aforementioned, symbolic and concolic testing tools are leveraged in automated software analysis to explore the program paths and reveal hidden vulnerabilities. Hence in this evaluation, we adopt a cutting-edge symbolic engine (KLEE [5]) to test the resilience of the obfuscated programs. Ideally program obfuscation brings in new challenges in reasoning path conditions, and hence would impede symbolic tools from finding new paths. In this evaluation, we use KLEE sample code [3] as the test case (the sample code is shown in Fig. 8).

KLEE could detect three paths in the original test case as expected. Actually based on different value of `x`, this program may traverse branches in which `x` equals 0, `x` is less than 0 or `x` is greater than 0, respectively. In contrast, we report KLEE could only reason **one** path condition for the obfuscated program. Due to limited information released by KLEE, we could not reveal the underlying reason that leads to the failure of the other two path conditions. Nevertheless, since Turing machine obfuscator makes the branch predicates more complicated, we envision that the internal constraint solver employed by KLEE is unable to yield a proper symbolic input which could "drill" into the branches protected

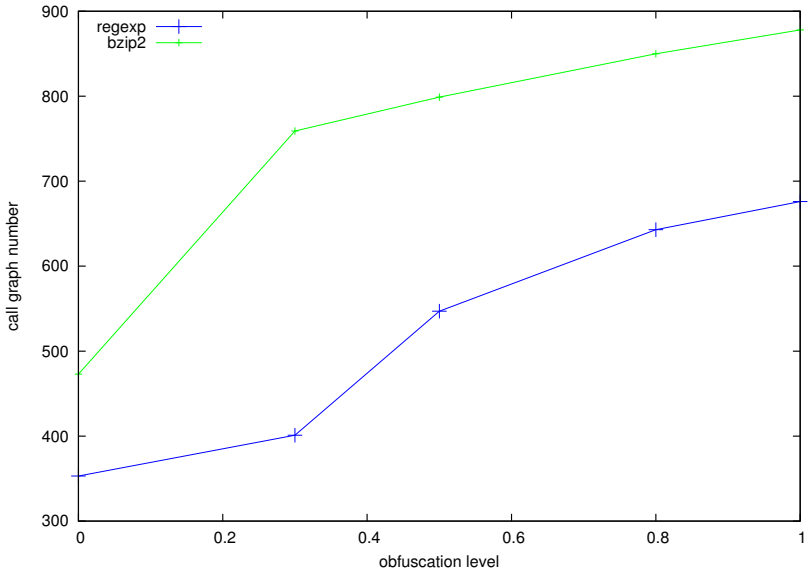

Fig. 7: Number of call graph edges in terms of different obfuscation levels.

```
1   int get_sign(int x) {
2     if (x == 0)
3       return 0;
4
5     if (x < 0)
6       return -1;
7     else
8       return 1;
9   }
10
11  int main() {
12    int a;
13    klee_make_symbolic(&a, sizeof(a), "a");
14    return get_sign(a);
15  }
```

Fig. 8: KLEE sample code used in our evaluation. All the path conditions are obfuscated.

by our tool. In sum, we interpret that Turing machine obfuscator can impede automated program analyzers from exploring the program paths.

5.4 Stealth

To evaluate the stealth of the obfuscated programs, existing work [23] propose to compare the instruction distributions of the original and obfuscated programs. If instruction distribution of the obfuscated program is distinguishable from its origin (e.g., `call` or `jmp` instruction proportions are abnormally high), it would

be an indicator that the program is manipulated. In this evaluation, we adopt this metric to measure the stealth of our Turing obfuscator.

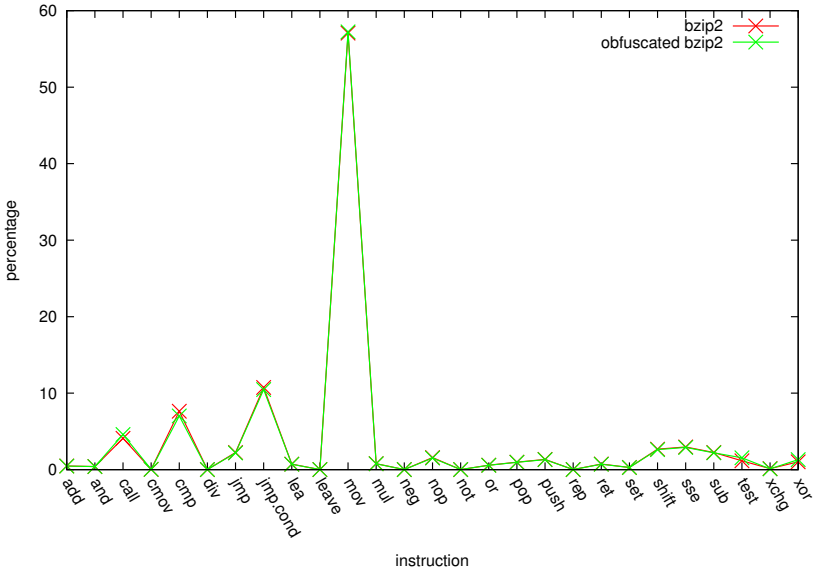

Fig. 9: BZIP2 instruction distribution comparison.

Consistent with previous research [23], we put assembly instructions into 27 different categories. Fig. 9 and Fig. 10 present the instruction distribution of the original and obfuscated programs (BZIP2 and REGEXP). Experimental 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

Performance penalty is another critical factor to evaluate 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 the cost.

In our evaluation, both original and obfuscated programs are executed on a server 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 calculate the average execution cost.

Fig. 11 presents the execution overhead results. For both cases, the execution time slowly grows with the increase of the obfuscation levels. As expected,

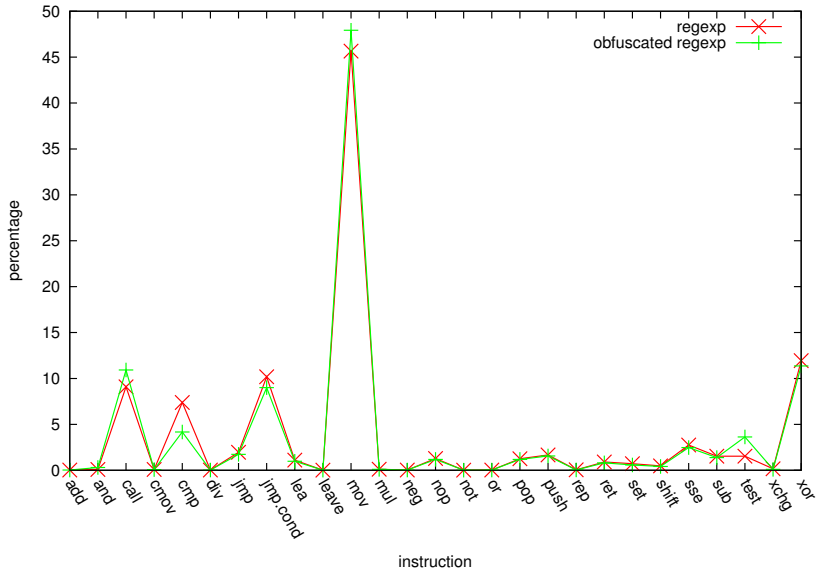

Fig. 10: REGEXP instruction distribution comparison.

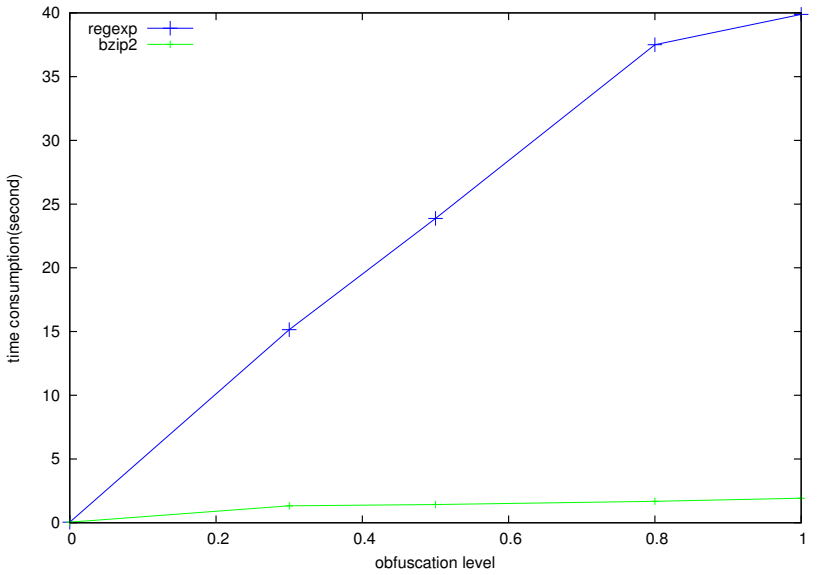

Fig. 11: Execution overhead in terms of different obfuscation levels.

program takes more time to execute when more instructions are obfuscated. Nevertheless, we interpret the overall execution overhead is still confined to a reasonable level. We also notice that there exists a difference between slopes of the two curves. Some further study on the source code shows 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.

6 Discussion

In this section we present the discussion of the proposed Turing machine obfuscation technique.

6.1 Complexity

In general, Turing machine model is a powerful but complex 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. Hence, it is hard—if possible at all—for adversaries with manual reverse engineering efforts to follow the calculation logic without understanding the transition table rules and state variables. In addition, automated binary analyzers (e.g., KLEE) can also be impeded due to the runtime complexity of a Turing machine. As shown in our resilience evaluation (§5.3), the constraint solver of KLEE failed to yield proper inputs to cover two of three execution paths.

To further improve the complexity, a promising direction is to perform "recursive" obfuscation towards the input program. That is, we employ the Turing obfuscator for the first round obfuscation, and further re-apply Turing obfuscator to obfuscate the Turing machine inserted in the first round. Existing work has pointed out that such "recursive" obfuscation approaches can usually improve the program complexity, while may also bring in non-negligible execution overhead [24]. We leave it as one future work to study practical strategies to recursively apply our technique for obfuscation.

6.2 Application Scope

Previous obfuscation work [21] 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 [23]. 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 software 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 semantics of the Turing machine code, return value of the obfuscator is indeed observable (since obfuscated branches are rewritten into function calls to the Turing obfuscator). Certain amount of information leakage may become feasible at this point.

We notice that existing work ([16, 15]) proposes a different approach at this step; control flow is directly guided (via `goto`) to the selected branch from their 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 [10]. 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 guarantees 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 serve as a double-edge sword; it impedes adversaries and potentially increases execution overhead as well. As reported in the cost evaluation (Fig. 11), we observed non-negligible performance penalty for both cases. One countermeasure here is to perform selective obfuscation; users can annotate sensitive program components for obfuscation. Such strategy would improve the overall execution speed without losing the major security guarantees.

7 Conclusion

In this paper, we propose a novel obfuscation technique using Turing machines. We have implemented a research prototype, Turing machine obfuscator, on the LLVM platform and evaluated on open source software with respect to functionality correctness, potency, resilience, stealth, and cost. The results indicate effectiveness and robustness of Turing machine obfuscation. We believe Turing machine obfuscation could be a promising and practical obfuscation tool to impede adversary analysis.

8 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. bzip2. `http://www.bzip.org`, 2017.
2. ida. `https://www.hex-rays.com/products/ida/`, 2017.
3. Klee sample. `http://klee.github.io/tutorials/testing-function/`, 2017.
4. slre. `https://github.com/cesanta/slre`, 2017.
5. 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.
6. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, 2006.
7. Haibo Chen, 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 (Micro '09)*, pages 391–400, 2009.
8. 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.
9. B. Jack Copeland. The church-turing thesis. Stanford encyclopedia of philosophy, 2002.
10. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, 2013.
11. Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.
12. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
13. Pengwei Lan, Pei Wang, Pei Wang, and Dinghao Wu. Lambda obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM'17)*, 2017.
14. 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.
15. 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.

16. 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.
17. Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
18. Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (USENIX Security '07)*, 2007.
19. Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, 2006.
20. Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '13)*, pages 263–272, 2005.
21. 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.
22. SingleTape. Turing machine. `http://turingmaschine.klickagent.ch/`, 2017.
23. 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.
24. 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.
25. Yan Wang. Obfuscation with Turing machine. Master's thesis, The Pennsylvania State University, 2017.
26. 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.
27. Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *Proceedings of the 16th European Conference on Research in Computer Security*, pages 210–226, 2011.
28. 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.
29. 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.