Obfuscation with Turing Machine

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

The Pennsylvania State University
University Park, PA 16802, USA
{ybw5084,szw175,pxw172,dwu}@ist.psu.edu

Abstract. Software security is a fundamental research domain in this threat emerging technology world. Control flow obfuscation is one of the important techniques to prevent hackers from understanding the program internal logic and leveraging software vulnerabilities to do damage. Hence, concealing important conditional branch logics are crucial for protecting software from being compromised. In this paper, we propose a novel control flow obfuscation method by leveraging the complexity of Turing machine. By entwining the original software programs with Turing machine execution, control transfers could be selectively obfuscated. Control flow graph and call graph could be significantly complicated. We implemented an obfuscation tool named Turing machine obfuscator with LLVM. Compared with previous works, our control flow obfuscation technique bears two distinct advantages. 1). Complexity: Complicated implementation of Turing machine makes it hard for attackers to follow the control flow graph logic. 2). Universality: Our obfuscation happens on intermediate representative level so the application scope is broadened to almost every language with an LLVM front-end compiler. We evaluate the obfuscator by potency, resilience, stealth, and cost, respectively. Experimental results show that Turing machine obfuscator could obfuscate programs in stealth with good performance and robustness.

Key words: software security, control flow obfuscation, Turing machine, LLVM

1 Introduction

Obfuscation derives from intellectual property protection. The Internet brings us unprecedented convenience along with idea plagiarism threat and copyright infringement. Concealing the algorithm of a software means a lot for the society especially for high-tech industry. Attackers could take advantage of the state-of-the-art techniques [22, 23, 24] to recover source code structure from binaries, exploit software vulnerabilities or to steal software ideas or algorithms. Obfuscation is mostly designed to impede such (malicious) reverse engineering process.

Recently, Software security research again draws people's attention because of infamous ransomware attack and severe vulnerabilities such as the "Wanncry" incidence and the OpenSSL heart bleeding bug. All of these malwares

exploit vulnerabilities inside a software. To exploit vulnerabilities in the software, attackers usually need to recover the program control-flow structures first. Dynamic code analysis is a popular and effective technique to explore inner logic of a program. *Concolic testing* is well-developed and widely-adopted technique which leverages both symbolic and concrete execution to cover all the program paths[1]. The concolic testing engines like SAGE[17] and KLEE[20] could yield new input which leads to a new execution path by solving path conditions as constraints based on previous execution. It has been proved effective in analysing control flow graph(CFG) of software[27].

Hence, a lot of anti-reverse engineering research has focused on preventing adversaries from analyzing important path conditions in a program[21, 5, 7, 8, 9]. 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 paper, we propose a novel control flow obfuscation method which leverages Turing machine to compute path conditions. *Church—Turing thesis* has shown that any function is computable by a human being without resource limitation concerns if and only if this function is computable by a Turing machine [25]. This means any functional component of software can be reimplemented as a Turing machine; the replaced code component and its corresponding semantic equivalent Turing machine is called *Turing Equivalent*.

In this work, we propose to simulate important branch conditions in original source code with its *Turing Equivalent* Turing machine. 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. A typical Turing machine leverages transition tables to guide the state transition, and such transition table-based execution would introduce additional computations to the target program. We believe the proposed Turing machine obfuscation would largely complicate the protected program, and also bring in new challenges for reverse engineering tasks.

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 path condition instructions for translation; the translated instruction will invoke its corresponding Turing machine component, which is semantically equivalent to the original path condition. After finishing the execution in the Turing machine "black box", the execution flow returns back to the original instruction, with a return value that indicates the branch selection. Inspired by previous work [4], we evaluate our obfuscator regarding four aspects which are potency, resilience, cost and stealth, respectively. Results indicate that Turing machine obfuscator could effectively obfuscate commonly-used software with acceptable performance penalty.

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 present discussion in section 6, and conclude the paper in Section 7.

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]) identify conditions that could trigger malware execution then using a hash function to transform such condition outputs. Hence, corresponding conditional codes which would be run with the trigger value were encrypted with a key generated based on the instruction trigger value. By this means the obfuscation analyzer could never get a chance to get the expected "launching code" consequently planted malware could never be executed. This technology works on certain fixed trigger value but not in scenarios that trigger values are intervals. This limitation narrows the application scopes greatly since a large volume of branch conditions are comparison operations. In addition, the encryption and decryption process in this methodology also introduce non-negligible overhead.

Popov et al. ([7]) propose to leverage signals ("traps") to replace the unconditional control transfer instructions such as `jmp` and `call` to impede disassembly operation which is the first step of reverse engineering. Moreover, dummy control transfers and junk instructions are also inserted after signal replacements. This method seems to be effective in fooling disassemblers but it cannot be applied in scenarios that the conditional control flow transfers need to be protected.

Another related work proposes to cover branch information 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, **?**]) take advantage of neural network to replace certain branch condition statements in source programs; the propose technique is evaluated to conceal conditional instructions and dynamic analysis such as concolic testing would be trapped to cover the protected branches. Although the idea is promising and the experimental results indicate the effectiveness to certain degree, in general neural network-based solution may not be suitable for such scenarios. 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 the selection of incorrect branches. Overall, neural networks not only introduce complexity but also unpredictability to the transformed programs. Trained models may behave very differently if given initial parameters with only some nuances, which means that it is very challenging to train an accurate enough model to simulate the conditional instruction. In addition, we notice that neural network usually consumes too much memory in the evaluations.

3 Turing machine obfuscation

3.1 Design Overview

In a program, a path condition statement compares two operands and selects a branch for control transfer based on the comparison result. In theory, Turing machine has been proved to be able to simulate the semantics of any computer algorithm. Hence, any program path 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 path condition statements in a program. The general workflow of a Turing machine obfuscated path condition statement is shown in Fig. 1. As shown in the figure, instead of directly computing a boolean value in the condition statement, we feed a Turing machine with the inputs (the value of operands) and let the Turing machine to simulate the execution.

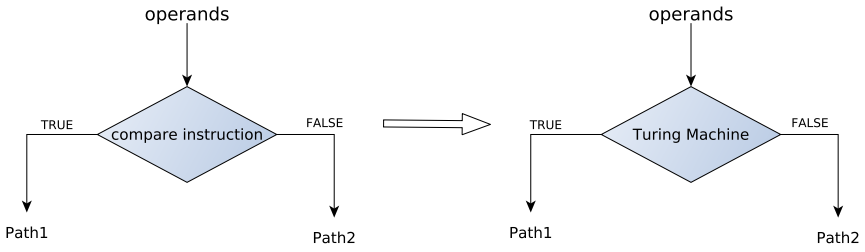

Fig. 1: Obfuscate a path condition statement through a Turing machine.

3.2 Turing Machine

Turing Machine is a mathematical model which could simulate any computer program. As shown in Fig. 2, a typical Turing machine consists of four components:

– An infinite-long tape which consists of a sequence of cells. Each cell holds a symbol defined in the tape alphabet. In implementation, Turing machine

obfuscator dynamically allocates new tape cells to construct an infinite tape used to store intermediate results.
- A tape head which could conduct read, write, move left and move right operations.
- A state register used to record the state of the Turing machine. Turing machine states are finite.
- A transition table that consists of all the transition rules defining how a Turing machine transfers from one state to another.

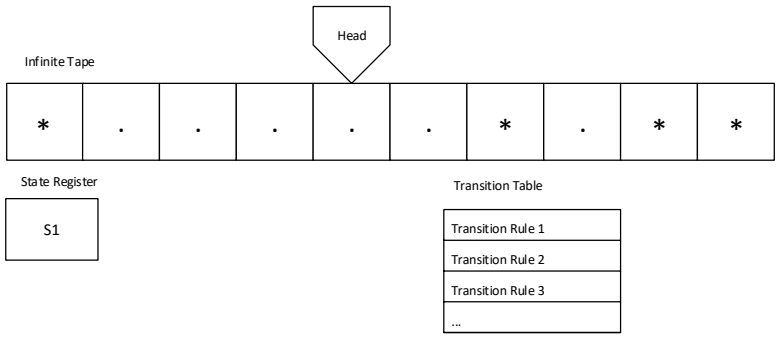

Fig. 2: Turing Machine Components

Transition Table Mathematically a transition rule could be represented by a 5 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 current tape cell
- D is the direction the head should move (i.e., "left" or "right")

Every 5 element tuple represents a transition table rule in 2. 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 program code and data. Hence, Turing machine is also deemed as the foundation of modern computer science development.

Turing machine initialization Initially, Turing machine is in "start" (S_0) state and tape records the Turing machine input. Blank symbol is symbolized as Char "*" on tape. Consistent with existing Turing machine simulator project [6], we use the length of "·" to represent an operand of integer type. For instance, integer 5 is represented as five continuous "·" on tape. Turing machine tape alphabet is $\{\cdot, *\}$. The tape in Fig. 2 displayed an initial state of Turing machine. The head of Turing machine is placed on the leftmost cell. Different integer operands are separated by a single blank symbol "*". Operands encoded on the tape of Fig. 2 are 5 and 1. When Turing machine starts to run, the head reads the current tape cell, combining with current state register value it could locate one and only one match transition rule in the transition table. This matched transition rule entry will direct Turing machine to change to next step.

Turing machine running 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 Turing machine to do simple algebra computations so it would 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 supports a Turing machine to conduct the `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 the design of "add" Turing machine which simulates the semantics of the addition operation. Other Turing machines(`substractiong` Turing machine, `multiplication` Turing machine and `division` Turing machine) used in this research are designed in a similar way. Through constructing this transition table, we essentially build rules which could concatenate two integer dot cells together. Take initial tape in Fig. 2 as an example. Following the rules in table 1, after a sequence of reading and writing operations based on the transition table, left operand 5 and right operand 1(enbodied by two series of operand dot cells) which are splited by *blank symbol*

"*" are merged into a long series of dot cells on tape; the length of outcome dot cells is the sum of two operands 6 as shown in Fig. 3.

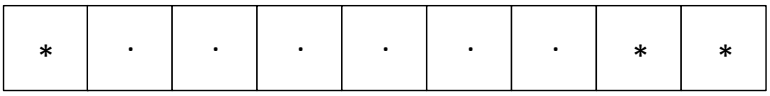

Fig. 3: Turing machine execution result

Adding Turing Machine Transition table encodes the running algorithm which is simplified in Algorithm 1. Algorithm 1 in fact states a method how to combine two series of dot cells on tape together. Following the algorithm, the isolator cell *blank symbol* is erased when Turing machine finally enter the "Halt" state.

Algorithm 1 Transition table algorithm for the "add" Turing machine.

```
1: procedure RUNNING PROCEDURE
2:     head ← blank cell before left operand starting cell
3:     while head != blank cell after right operand do move right
4:     move left
5:     last dot cell of right operand ← blank symbol
6:     while head != blank cell isolator of two operands do move left
7:     blank cell isolator ← dot
8:     while head != blank cell befor left operand do move left
9:     Halt;
```

‘

Turing Machine of other operations As previously discussed, given any program algorithm, there must exist a corresponding Turing machine. Since our Turing machine obfuscator concentrates on obfuscating conditional branch instruction with arithmetic operations(*e.g.* `if(a*b>c)...`), Turing machine obfuscator need to contain transition tables which is capable of doing arithmetic operations such as $+, -, \times, \div$.

To implement the arithmetic operations, besides the addition transition table shown in Table 1, we construct three more tables for subtraction, multiplication and division operations. Their transition tables are relatively more complex than Table 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 $\leq, \geq, \neq$ between operands, we take advantage of subtraction Turing machine to calcu-

lated the difference of them. Based on calculation result, True/False 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 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 distinct transition rules. A "add" Turing machine could not represent "subtract" operations. A Turing machine encoded with 2 + 3 could not conduct addtion operation 5 + 6. To some degree, each Turing machine is encoded with a fixed algorithm and input data. However, in non-trivial programs, path condition operands could be any kind of operation such as $x+5$ or 6×7. Many of these algebra expressions would correspond to different Turing machines. Hence, we need an unified translator to represent arbitrary computations. Universal Turing machine is designed to simulate arbitrary Turing machines. It stores and interprets arbitrary a Turing machine on a single tape. As shown in Fig. 4, both input data and transition table are written on the tape of the universal Turing machine. In some sense, a universal Turing machine acts as the interface for us to employ Turing machines of different semantics.

Universal Turing machine bears the essence of the modern computer which is being programmable. Through storing different transition tables on a sequence of cells on the tape, an universal Turing machine can actually perform semantic equivalent computation to arbitrary programs; as aforementioned, such Universal Turing Machine and the replaced algebra expression are *Turing Equivalent*. In our Turing machine obfuscator, we can invoke a Universal Turing machine function call on interested conditional control transfer instructions and related arithmetic operation instructions.

Tape construction Before universal Turing machine execution, a tape which contains the input operands and Turing machine transition tables is generated. For each operation of "+", "−", "×" and "÷", corresponding transition table is also embedded on tape. As aforementioned, integer inputs are encoded as strings of "." with different lengths. Following a universalized tape formatting protocol across Turing machine obfuscator, we concatenate transition tables and inputs to form a tape for the universal Turing machine. In this way, all the information needed for computations exists on the tape. During the execution of universal Turing machine, it would first extract a proper transition table then perform one step computation.

Negative Integer Operand Preprocess still not clear In software programs, integer operands comprise both positive and negative cases. Turing machine could only dispose of positive operands in consequence of we use the length of dot cell on tape to represent a integer. This means we have to preprocess the invalid operands in Turing machine obfuscator. In the preprocessing stage,

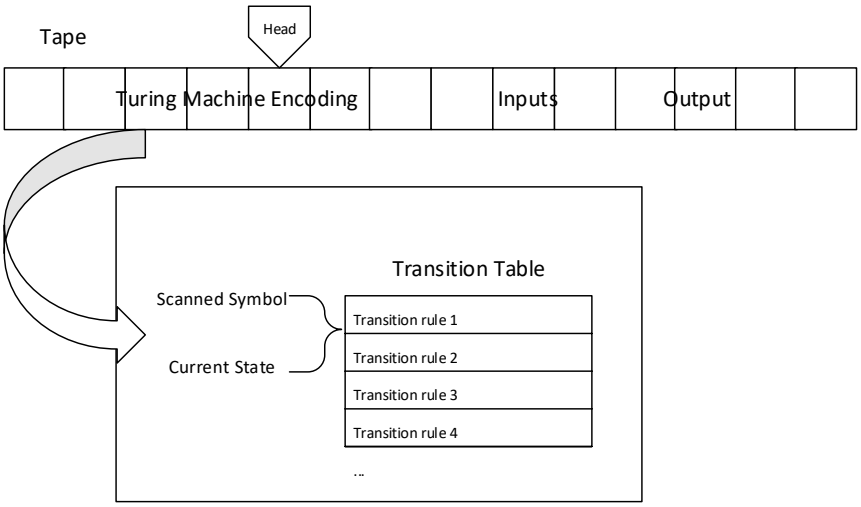

Fig. 4: Overall workflow of a universal Turing machine.

Turing machine obfuscator could convert invalid operands to its opposite number to run Turing machine. Calculate result from Turing machine is also revised before returning. For instance, $-4 + (-6)$ is preprocessed to $4 + 6$, afterwards -10 which is the opposite number of 10 is returned. $4 + (-6)$ is preprocessed to Turing machine operation $6 - 4$, and the opposite number of the outcome 2 (i.e., -2) is returned. In preprocess stage, any arithmetic operation would be transformed to a valid integer operation with $+$, $-$, $\times$, $\div$ for Turing machine.

4 Implementation

Our proposed technique is implemented as a transformation pass of the LLVM compiler suite [19]; the LLVM intermediate representation (IR) is obfuscated and then compiled into binary executable. 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 obfuscator's source code into LLVM IR. The obfuscator then iterates IR instructions to identify obfuscation candidates and perform the transformation. The instrumented IR codes are further compiled into the executable file in the final step. We implement the Turing machine obfuscator in C, with in total XXX lines of code.

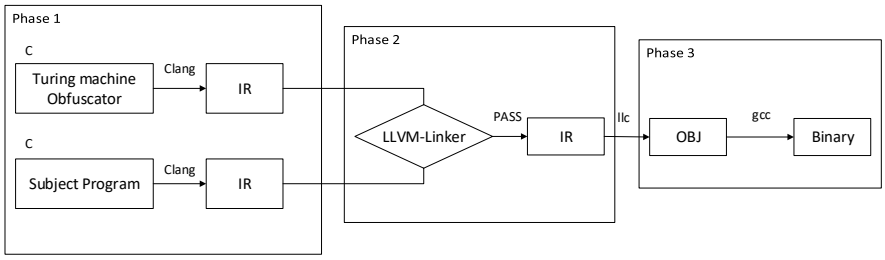

Fig. 5: Workflow of Turing machine obfuscator.

4.1 Phase One: Translating Source Code to IR

As aforementioned, we first compiles 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 LLVM IR, this IR-based implementation could broaden the application scope our tool comparing with previous work [2, 8, **?**]. On the other hand, 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

During the static analysis pass, our Turing machine obfuscator locates all transformation candidates on the IR instructions. The LLVM Pass framework is a core module of the LLVM compiler suite to conduct analysis, transformation and optimization during the compile time [26]. We build a extra pass within this framework to iterate and analyze every IR instruction in each module of the input program.

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 instructions to compute different branch predicates (in total 10 kinds) such as: equal, not equal, less than, greater than. integer predicate

4.3 Phase Three: Transformation Pass

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 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. The essence of Turing obfuscator is to hide program control transfers; 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.

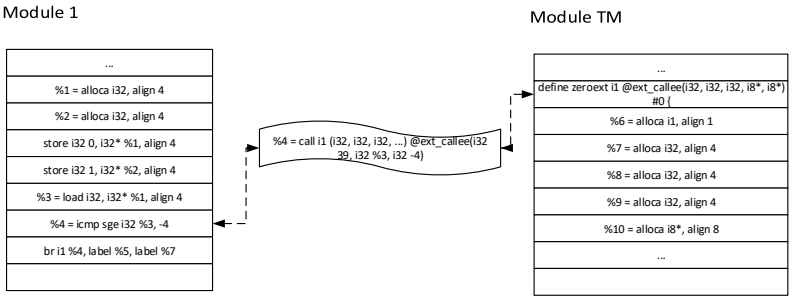

Fig. 6: Phase Three: Transformation Pass

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, **?**]) leverages a cross-procedure jump at this step; an indirect jump from the black box of the Turing machine to a selected branch. We present further discussion on both branch determination strategies in § 6.

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 § 4.1).

5 Evaluation

Inspired by previous research [4, 2, **?**], 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, 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 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 [10] and regular expression engine slre [11]. In Turing machine obfuscator, we collect all path conditional branch instructions (e.g., `icmp`). Through *Use-Define chain*, we also locate all instructions that determine the value of the conditional branch instruction, all these instructions are deemed as transformation candidates. Obfuscation level is an index which represents the ratio between obfuscated instructions and all 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 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] which is a well-known commercial disassembler and debugger, we recover CFG and call graph information from both original and obfuscated binaries. By traversing the graph, 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 2. 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 cyclomatic number and knot number (these two metrics are introduced in [14, 15]). Cyclomatic metric is defined as

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

$$Cyclomatic = E - N + 2$$

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 weighs how complicated a program is in terms of logic diversion number. Results in Table 3 shows that both knot number and cyclomatic number notably increase after Turing machine obfuscation. Overall, we interpret Table 2 and Table 3 as promising results to show program becomes more complex after the Turing machine obfuscation.

Table 3: 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

Besides picking 50% as the obfuscation level in evaluating potency, we also conduct experiments with obfuscation levels as 30%, 80% and 100%. Fig. 7 shows the number of call graph edges regarding different obfuscation levels. Observation shows that with a higher obfuscation level, the number of call graph edge increases. We interpret the results that the obfuscated program become more complicated with the obfuscation level increases.

5.2 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 inner logic control flow graph[28, 1, 27]. KLEE [20] is static analysis tool based on LLVM platform and it could generate enough test cases to ensure a code coverage threshhold. With the help of KLEE, it would be easy to conduct automated deobfuscation by concolic testing. We chose KLEE as the deobfuscation tool to evaluation resilience of Turing Machine obfuscator. We used a piece of sample code from KLEE [20] as the subject program (the sample code is shown

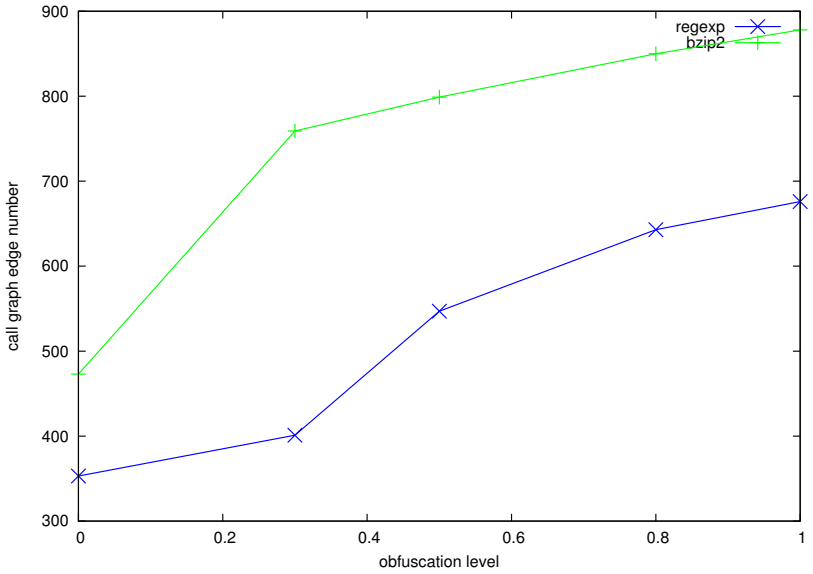

Fig. 7: Number of Call graph edges in terms of different obfuscation levels.

in Fig. 8). The subject program need to be converted to IR codes since KLEE works on IR level.

```
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.

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, repectively. In contrast, after subject program is obfuscated by our Turing machine obfuscator, 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 the failure of KLEE. Since Turing machine obfuscator make the conditional branches more complicated, our guess is that the internal constraint solver employed by KLEE is unable to yield proper symbolic input which "drill" into the branches protected by our Turing machine obfuscator.

5.3 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, [3] compares the instruction distributions of 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 obvious indicator that the program is manipulated. We adopted this metric to evaluate our Turing obfuscator. Obfuscation level for stealth evaluation is set to 50%.

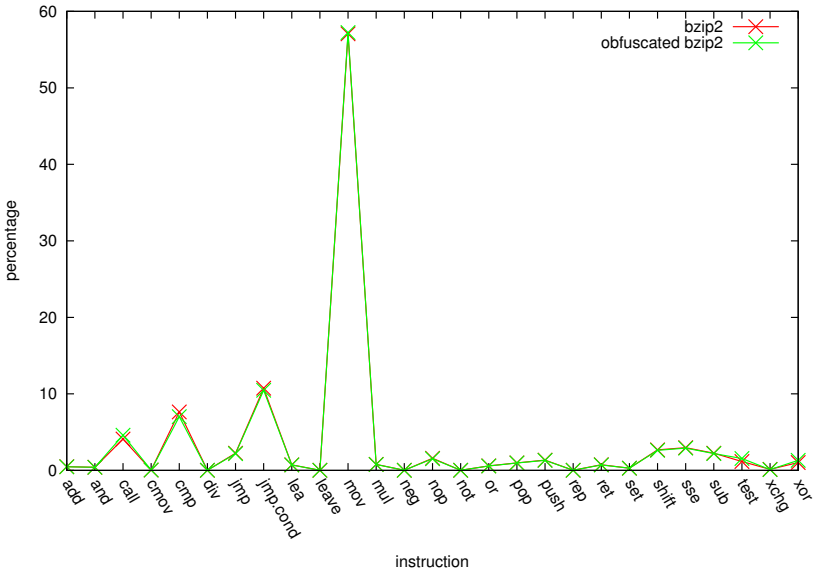

Fig. 9: BZIP2 instruction distribution comparison

Consistent with previous research ([3]), 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). Experiment results indicate that the instruction distribution after obfuscation is very close to the origin

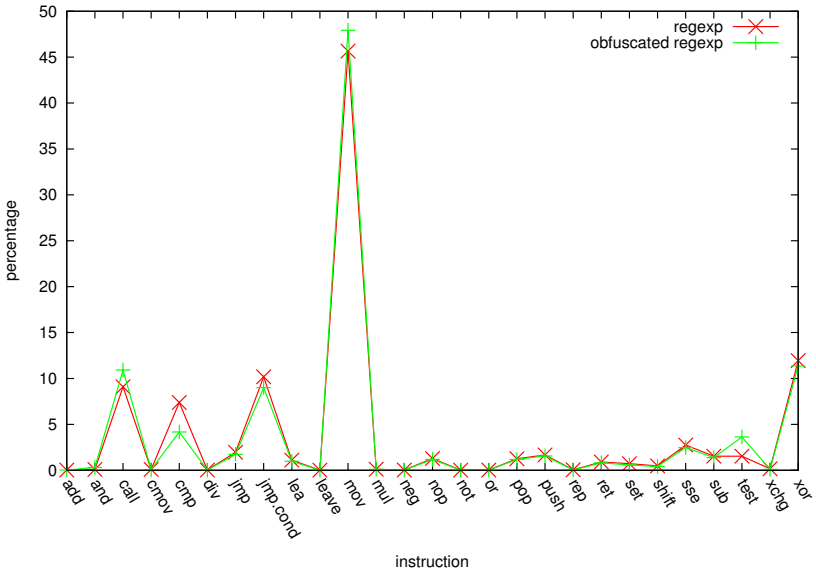

Fig. 10: regexp instruction distribution comparison

distribution. Comparing these two figures together, we noticed that instruction diststribution variance of REGEXP is bigger than BZIP2, it makes sense since REGEXP consists of 8117 lines of Code and REGEXP contains only 1391 lines of C code. In sum, small instruction distribution variation is a promising result to show the proposed technique would obfuscate programs in a stealthy way.

5.4 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 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 SLRE runs 149 test cases provided by its shipped test cases. We ran each program three times and recorded the average time cost as the final result.

Fig. 11 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

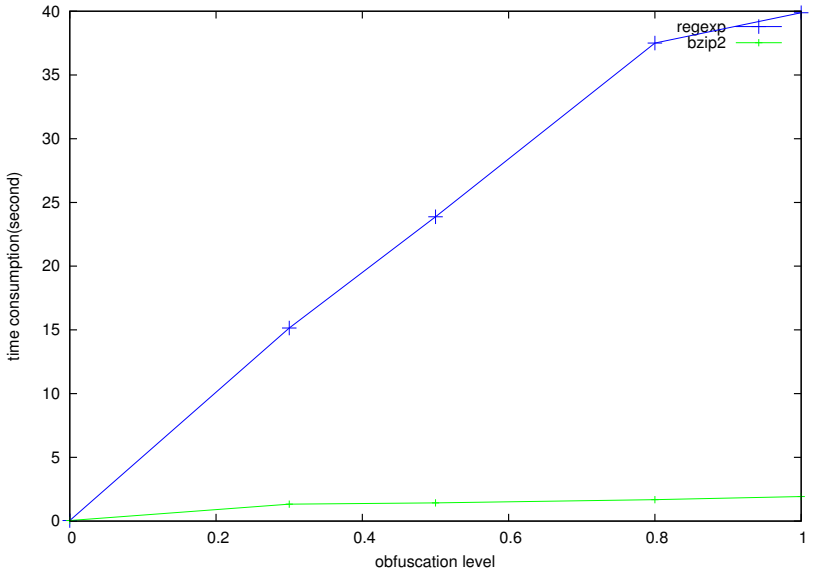

Fig. 11: Execution overhead in terms of different obfuscation levels.

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, the regexp program employs more recursive calls than BZIP2, thus may contribute to the relatively higher cost grow as well.

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 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" or "write" 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, the intrinsic Turing machine complexity can also defeat automated deobfuscation tools (e.g., KLEE). As reported in our resilience evaluation (§ 5.2),

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 functionality 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, **?**]) proposes a different approach at this step; control flow is directly guided (via `goto`) to the selected branch from the Neural network obfuscator. While intuitively this approach seems to hide the explicit return value, such approach 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 [29]. Suppose we model a branch predicate with a Turing machine function (e.g., TM), the general idea is to further transform TM into a matrix function, 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 is a double edge sword; it impedes adversaries and increases execution overhead at the same time. As reported in the cost evaluation (Fig. 11), we observed non-negligible performance

penalty for both cases. One solution is to perform selective obfuscation; users can mark sensitive program codes and Turing machine obfuscator would only harden those parts. Such strategy would improve the overall execution speed without sacrificing the major security requirements.

7 Conclusion

In this paper we proposed a novel obfuscation technique taking advantage of the merits of Turing machine execution. We designed a prototype called Turing machine obfuscator will LLVM platform. We implmented Turing machine and universal Turing machine for arithmetic operators $+, -, *$ and $\div$. Return value of Universal Turing machine execution is used to direct control flow. We evaluated Turing machine obfuscator with respect to pontency, resilience, stealth and cost with two open source software as subjects. Evaluation results indicated effectiveness and robustness of Turing machine obfuscator. We believe Turing machine obfuscator could be a solid and pratical obfuscation tool to protect software security.

References

1. Sen, Koushik, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 5. ACM, 2005.
2. Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao and Chunfu Jia. "Control flow obfuscation using neural network to fight concolic testing." International Conference on Security and Privacy in Communication Systems. Springer International Publishing, 2014.
3. Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, Dinghao Wu. "Translingual obfuscation." Security and Privacy (EuroS&P), 2016 IEEE European Symposium on. IEEE, 2016.
4. Collberg, Christian, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs." Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1998.
5. Sharif, Monirul I., et al. "Impeding Malware Analysis Using Conditional Code Obfuscation." NDSS. 2008.
6. `http://turingmaschine.klickagent.ch/`
7. Popov, Igor V., Saumya K. Debray, and Gregory R. Andrews. "Binary Obfuscation Using Signals." Usenix Security. 2007.
8. Zhi Wang, Chunfu Jia, Min Liu. "Branch obfuscation using code mobility and signal." Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual. IEEE, 2012.
9. Zhi Wang, Jiang Ming, Chunfu Jia, Debin Gao. "Linear obfuscation to combat symbolic execution." European Symposium on Research in Computer Security. Springer Berlin Heidelberg, 2011.
10. `http://www.bzip.org/`
11. `https://github.com/cesanta/slre`

12. Chen, Haibo, et al. "Control flow obfuscation with information flow tracking." Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. 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. https://en.wikipedia.org/wiki/Turing_machine
17. Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." NDSS. Vol. 8. 2008.
18. https://en.wikipedia.org/wiki/Universal_Turing_machine#/media/File:Universal_Turing_machine.svg
19. https://en.wikipedia.org/wiki/LLVM
20. https://klee.github.io/
21. Xu, Dongpeng, Jiang Ming, and Dinghao Wu. "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method." International Conference on Information Security. Springer International Publishing, 2016.
22. Ming, Jiang, et al. "Loop: Logic-oriented opaque predicate detection in obfuscated binary code." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.
23. Lee, Gareth, et al. "Using symbolic execution to guide test generation." Software Testing, Verification and Reliability 15.1 (2005): 41-61.
24. Molnar, David, Xue Cong Li, and David Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs." USENIX Security Symposium. Vol. 9. 2009.
25. https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis
26. http://llvm.org/docs/WritingAnLLVMPass.html
27. Sen, Koushik, and Gul Agha. "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools." International Conference on Computer Aided Verification. Springer Berlin Heidelberg, 2006.
28. Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." The 11st USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2008.
29. Garg, Sanjam and Gentry, Craig and Halevi, Shai and Raykova, Mariana and Sahai, Amit and Waters, Brent. "Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits." The 53rd Annual Symposium on Foundations of Computer Science (FOCS). 2013.
30. Hanoyu Ma, Ruiqi Li, Xiaoxu Yu, Chunfu Jia, Debin Gao. "Integrated Software Fingerprinting via Neural-Network-Based Control Flow Obfuscation." IEEE Transactions on Information Forensics & Security (TIFS). 2016.