Obfuscation with Turing Machine

Yan Wang[1], Dinghao Wu,[2] Shuai Wang, and Pei Wang

Pennsylvania State University, State College,PA 16801, USA,
`ybw5084@ist.psu.edu`

Abstract. Software security is a fundamental research domain in this threat emerging technology world. Control flow obfuscation one of 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, software control flow graph and call graph could be significantly complicated. We implemented an obfuscation tool named Turing machine obfuscator with LLVM. We evaluate the obfuscator by potency, resilience, stealth and cost respectively. Experiment 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. Reverse engineering is often used by Hackers to recover source code structure from binaries, analyze software vulnerabilities or to steal software ideas or algorithms. Obfuscation is a technique to block or harden the process of reverse engineering.

Recently, Software security has become a bigger and bigger concern in research community because of infamous ransomware attack and severe vulnerability such as the recent "Wanncry" incidence and the OpenSSL heart bleeding bug.This incidences challenge the computer world greatly. Hackers endeavors to figure out venerabilities inside a software program. Knowing the software architecture and program logic like control flow graph is an indispensable prerequisite for hackers to analyze the original codes. With the help of some monitoring techniques hackers could even iterate all possible paths to try to restore all important branch information along the execution paths. *Concolic testing* is an example which exploits symbolic execution given a certain input while it keeps changing input data until the code coverage proceed a threshhold[1] as shown in Figure

1. It has been proved to work in restoring the branch information in the original source codes. Hence, a lot of research focus on preventing bad guys from figuring out the essential logic. Hiding important “crossroad” in a source program. Control flow obfuscation is one of these techniques. Control flow obfuscation aims at hiding conditional transfers and complicating the execution path within a source program. Through replacing or inserting extra control flow graph edges, the original software logic becomes harder or even impossible to trace. Previous research[2] have proved the effectiveness of control flow obfuscation.

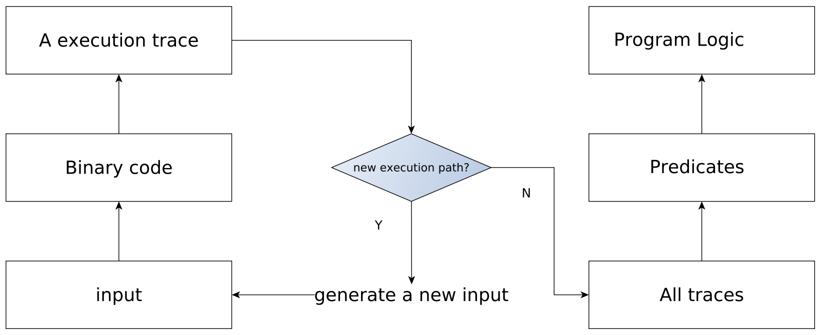


Fig. 1: Reverse engineering with concolic testing

In this paper, we propose a novel control flow obfuscation method with Turing machine embedded in the original program source codes. Turing machine is the essence of computation so it could calculate accurately all kinds of computer operation. In this way, important branch conditions in original source code could be replaced by a Turing machine execution which yields the very identical conditional value. Besides, a Turing machine behaves like a state machine so it contains a lot of branch condition checks in Turing machine transition table. In this whole process, sensitive branch information could be successfully hidden in Turing machine call graphs. In the meantime, Turing machine execution greatly introduces computational complexity to the target program so it is almost impossible to read the obfuscated programs to do reverse engineering.

At the current stage, we only obfuscated conditional transfer instructions with integer operands. Utilizing LLVM, we could turn original program source codes into intermediate the representation(IR). Turing machine obfuscator takes over from IR and selects interested instruction candidates which would be redirected to semantic equivalent Turing machine execution. After iterating the call graph in the Turing machine “black box”, the original call graph resumes from the obfuscator interrupt and the whole program control flow graph is greatly expanded and complicated. Since LLVM is the implementation foundation, currently our proposed obfuscator could only be applied to C/C++ source programs. Inspired by previous works[4], we evaluate our obfuscator from four dimensions which are potency, resilience, cost and stealth respectively. Results indicate that Turing machine obfuscator could effectively obfuscate target source codes with acceptable cost overhead.

This paper is organized as follows. Section 2 discusses related works on obfuscation, especially control flow obfuscation. Section 3 illustrate the idea and architecture of Turing machine obfuscator. Obfuscator implementation is discussed in section 4. Section 5 discusses the evaluation result of our proposed obfuscator. Finally, we conclude this paper in section 6.

2 Related Works

Generally speaking, reverse engineering is divided into static analysis such and dynamic analysis. To battle static reverse engineering, researchers usually focus on hardening disassembling and decompiling process. To combat the dynamic reverse engineering techniques such concolic testing, conditional transfer logic must be hidden from adversaries. Control flow obfuscation has been proved effective in previous works.

In [5], the authors identified conditions that could trigger malware execution then using a hash function to transform the values which could launch malware. Afterward, correspondent 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 such as $>$ and $<$. This limitation narrows the applicable conditions greatly since a large volume of branch conditions are comparison operations. In addition, the encryption and decryption process in this methodology also introduce nonnegligible overhead.

In [6], the authors used signals(“traps”) to replace the control transfers unconditional instructions like “jmp” and “call” in order to confuse disassembly operation which is the first step of reverse engineering. Dummy control transfers and junk instructions were also inserted after signal replacements. This method seems to be effective in fooling disassemblers but it can’t be applied in scenarios that conditional instruction logic needs to be protected from being figured out.

In [7], the authors tried to cover branch information leveraging a remote trusted third party environment. The idea seemed to work while it not only introduced a great network overhead but also rely on trusted network accessibility which can’t be guaranteed in reality. This drawback means this idea can’t be applied in common obfuscation cases.

In [2], the authors took advantage of neural network to replace appropriate conditional instructions in source programs in order to achieve the goal of consealing conditional instruction logic thus dynamic analysis like concolic testing could never dig useful branch information. Although the idea looks promising and the results indicates the effectiveness of this methodology to some degree, fundamentally we don’t believe neural networks solution is suitable for such scenarios. To the best of our knowledge, neural networks work like a black box.

It lacks the rigorous mathematical proof to illustrate a correct result must be generated given an input. Neural networks not only introduced complexity but also unpredictability to the original source programs. Trained models may behave very differently if given initial parameters with only some nuances, which means that it is also very hard to train an accurate enough model to simulate the conditional instruction. In addition, we noticed that neural networks consume too much memory in the evaluation experiments.

3 Turing machine obfuscation

In a computer program, conditional transfer instruction compares two operands and direct the following running path based on the comparison result. In theory, Turing machine has been proved to be able to simulate any algorithm and generate a corresponding correct answer. If a problem could be solved mathematically, it could definitely be solved by a Turing Machine. Taking advantage of Turing machine's powerful computational ability, complexity and mathematical correctness, we introduce Turing machine to the control flow obfuscation process. Through Turing machine execution, the identical boolean branching signal could be produced to guide further execution direction and the conditional logic is protected from being discovered. The intuitive methodology of Turing machine conditional branch transformation is shown in Figure 2.

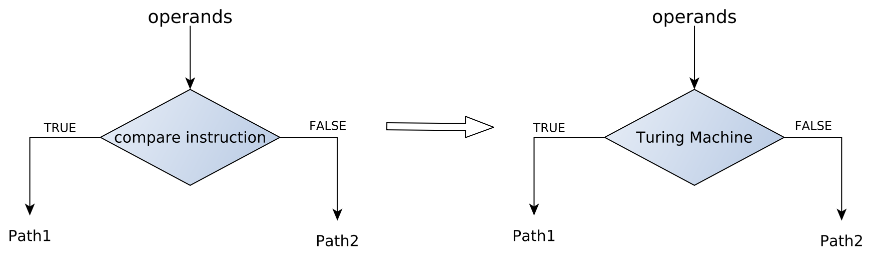


Fig. 2: Turing Machine Obfuscation Overview

3.1 Turing Machine

Turing Machine is a mathematical model which could operate any computer program algorithm through reading and writing a infinite tape. Turing machine contains four components shown as Figure 4[14]:

– An infinite tape composed of cells. Each cell holds a symbol defined in the tape alphabet.

– 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 denoting how Turing machine transfer from one state to another.

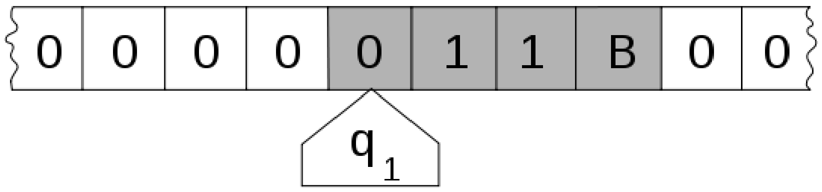


Fig. 3: Turing Machine Components[14]

Formally a Turing machine could be represented by a 7 element tuple[14]:

$$(Q, \mathit{\Gamma}, b, \mathit{\Sigma}, \delta, q_0, F)$$

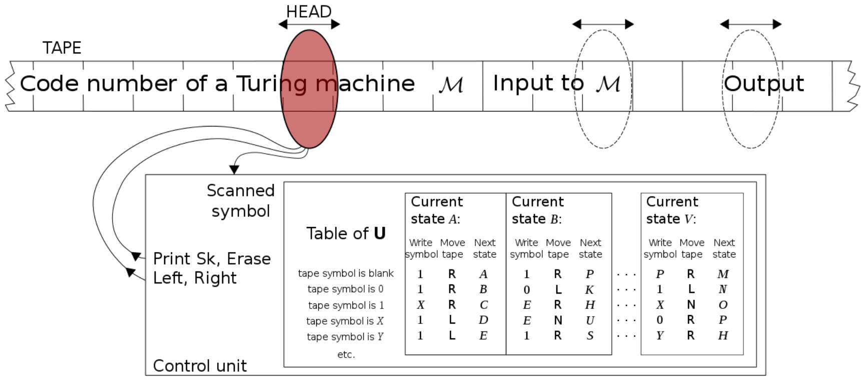
where:

– Q is finite state set
– $\mathit{\Gamma}$ is finite tape symbol set named alphabet
– b is the tape blank symbol. $b \in \mathit{\Gamma}$
– $\mathit{\Sigma} \subset \mathit{\Gamma} \backslash \{b\}$ is the actual input symbol set without blank symbol
– δ is the transition table composed of transition rules in form of $(Q \backslash F) \times \mathit{\Gamma} \rightarrow Q \times \mathit{\Gamma} \times \{L, R\}$
– $q_0 \in Q$ is the initial state
– F is finite acceptable states set. $F \subset Q$

Given any program algorithm, there must be a corresponding Turing machine simulation. Since our Turing machine obfuscator concentrate on integer control transfer instruction replacement, Turing machine obfuscator contains transition tables which could do arithmatic operations like $+, -, \times, \div$ and all compare operations like $\leq, \geq, \neq$ *etc.* Arithmatic operation result is displayed on tape when Turing machine reached the "accept" state. Compare operation result is 0 or 1 returned by Turing machine obfuscator.

3.2 Universal Turing machine

Although Turing machine could perform powerful and rigorous algorithm simulation, its inner logic is fixed and determined by the transition table. For instance, a Turing machine capable of doing adding operation could only simulate "add" operations. To some degree, every Turing machine is encoded with a internal fixed algorithm. However, in practical programs, control transfer could be any kinds of operation. Consequently, we need an algorithm translator to interpret the transformation rules of evey Turing machine. Universal Turing machine is used to store the inner algorithm logic of a Turing machine on a single tape. As shown in Figure 4, both the input and the transition table are written on a single universal Turing machine tape. In some sense, universal Turing machine is the interface for us to utilize the strong computability of a certain Turing machine.



	Current state A:			Current state B:				Current state V:		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state		Write symbol	Move tape	Next state
tape symbol is blank	1	R	A	1	R	P	···	P	R	M
tape symbol is 0	1	R	B	0	L	K	···	1	L	N
tape symbol is 1	X	R	C	E	R	H	···	X	N	O
tape symbol is X	1	L	D	E	N	U	···	0	R	P
tape symbol is Y	1	L	E	1	R	S	···	Y	R	H
etc.										

Fig. 4: Universal Turing Machine [15]

Universal Tuing machine bears the essence of modern computer which is being programmable. Through storing the transition table on the first several cells on the tape, an universal Turing machine actually loaded a piece of program which could yield identical results with the control transfer we could replace with. In our Turing machine obfuscator, we invoke a Universal Turing machine function call on interested control transfer instructions.

4 Implementation

We implemented Turing Obfuscator on Intermediate Representation(IR) level with the help of LLVM[16]. In general Turing obfuscator employed three phases to generated target obfuscated binary as shown in Figure 5. The first is to compile both target source program and obfuscator source code to IR, then the obfuscator processes IR instructions with several passes. At last these instrumented IR codes are compiled again and linked to the final obfuscated binary. Turing machine obfuscator is implemented with C.

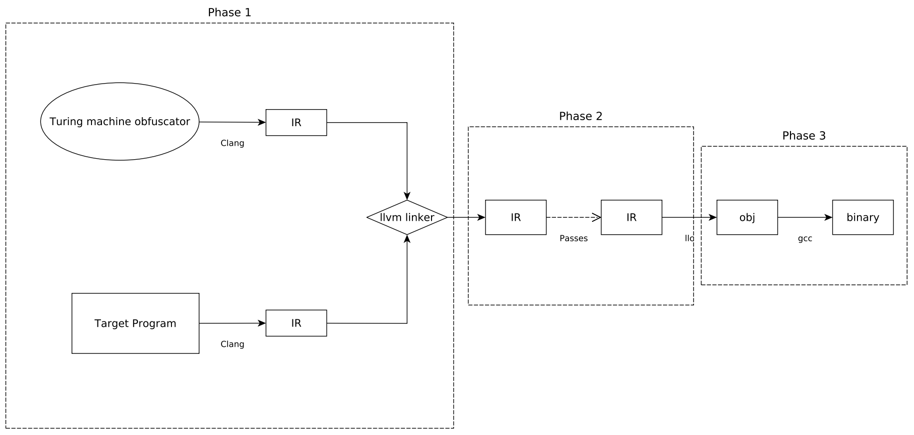


Fig. 5: Turing Mahine Obfuscator work flow

4.1 Translating Source code to IR

Turing machine obfuscator converts target source program to LLVM IR in phase 1. The reason behind this is LLVM could generalize our Turing machine obfuscator in consequence of the abundance of LLVM front end compilers. LLVM provides many front end compilers which could turn source programs written in C, C++, Python and other languages into LLVM IR. Since obfuscation transformation takes place in IR level, the applicable range of Turing obfuscator is significantly broadened compared to previous works[2],[7]. In experiments, we use software programs in C as target source codes so Clang-5.0 is the front end compiler.

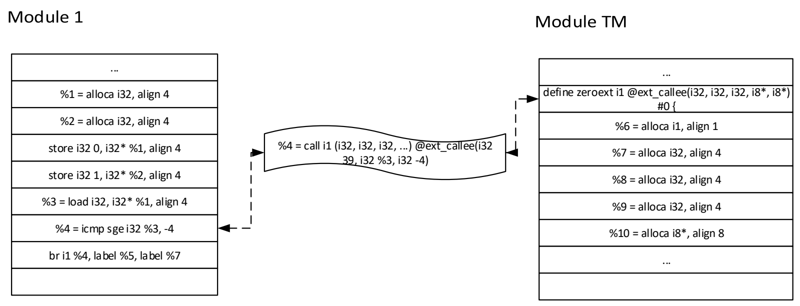
4.2 Static Analysis Pass

After preparation phase 1, Turing machine obfuscator does some static analysis on source program IR to locate all transformation candidates. LLVM Pass framework is a core module of LLVM to conduct instruction granularity analysis, transformation or other compiler level optimization. We built a Pass with this framework to iterate and analyze every IR instruction in each module of a source program. Instructions of Turing machine are filtered out and prevented from being obfuscated to avoid endless recursion. Currently we only extract control transfer instructions with integer operands as obfuscation candidates. These candidates comprise 10 kinds of instructions including : equal, not equal, unsigned greater than, unsigned greater or equal, unsigned less than, unsigned less or equal, signed greater than, signed greater or equal, signed less than, signed less or equal.

4.3 Transformation Pass

After the static analysis Pass we got all eligible instruction candidates. Afterwards, we built another Transformation pass to transform some or all of these candidate control transfers to a function call to universal Turing machine interface. Result of universial Turing machine call is stored to a register inside LLVM which is used to direct following path label.The essence of Turing obfuscator is to hide important control tranfers inside IR codes. Turing machine obfuscator could obfuscate any designated control transfer inside a target program. In experiments we randomly select a portion of the candidates and do the transformation.

We noticed that if Obfuscation could cover branch information further if we could jump back to a caller function basic block based on the Turing machine running result like in [2], but essentially it still expose branch trace since we need to record the basic block addresses of each branch. To the best of our knowledge, perfecting branch hiding technique does not exist since there is always a way to figure out inner logic of a certain program given enough time and energy. We need to harden this process as much as possible. Currently we adopt the IR replacement way to invoke Turing machine.



```
...
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 1, i32* %2, align 4
%3 = load i32, i32* %1, align 4
%4 = icmp sge i32 %3, -4
br i1 %4, label %5, label %7
```

```
%4 = call i1 (i32, i32, i32, ...) @ext_callee(i32
39, i32 %3, i32 -4)
```

```
...
define zeroext i1 @ext_callee(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
...
```

Fig. 6: Transformation Pass

5 Evaluation

Inspired by previous work[4], we evaluated our Turing machine obfuscator based on four metrics which are *potency*, *resilience*, *stealth* and *cost* respectively. Potency weighs the complexity of a program in order to show how competent an obfuscator is. A good obfuscator also need to protect itself from being deobfuscated by tools. To measure how well an obfuscated program could withstand automatic deobfuscation techniques, metric resilience is introduced. Besides automated deobfuscators, a lot of reverse engineering works are conducted by hackers, obfuscated program should not be too different from the original one or it would be easy for adversaries to recognize. We used stealth to measure to how well a obfuscated program resembles the original one. Cost is naturally employed to measure computing overhead of a software program. Obfuscation would inevitably induce more overhead while the cost should be constricted to an acceptable level.

We choose two popular open source programs as target programs to verify the effectiveness of our Turing machine obfuscator: compress tool bzip2[8] and regular expression engine slre[9]. In Turing machine obfuscator, we filtered out all integer conditional transfer instruction as replacement candidates. Obfuscation level is an index which means the ratio between obfuscated instructions and all instruction candidates. In our experiments, we arbitrarily set it to 50% which means half of all conditional transfer candidates are randomly chosen and obfuscated.

5.1 Potency

Control flow graph(CFG) and call graph provide useful information about the general structure of a program so they are the traditional foundation for static software analysis. With the help of IDA Pro[11] which is a well-known commercial disassembler and debuger, we extracted CFG and call graph from original and obfuscated binaries. Through analyzing both graphs, we figured out basic block number, call graph edge number and control graph edge number. These static metrics are also used to indicate the complexity of a target program in previous work[10]. Analysis result are shown in table 1. Comparing the metrics of source program and corresponding Turing machine obfuscated program, we found program complexity is strengthened in terms of each metric.

Program	# of CFG Edges	# of Basic Blocks	# of Function
bzip2	4283	2837	78
obfuscated bzip2	4195	2828	134
regexp	906	619	25
obfuscated regexp	1122	773	43

Table 1

Besides the traditional static evaluation, we refer to [12] and [13] to further quantify Turing machine obfuscated programs. The cyclomatic number and knot number were introduced in these works. Cyclomatic metric is defined as

$$Cyclomatic = E - N + 2$$

where E and N respectively represent the number of edges and the number of nodes in a CFG. Knot number means the amount of edge crossings in the CFG. These two metrics intuitively weighs how complicated a program is in terms of logic diversion number. Results in Table 2 show that both knot number and cyclomatic number increase after Turing machine obfuscation. Therefore, the potency of Turing machine obfuscator is proved.

Program	# of Cyclomatic	# of Knot
bzip2	1448	11982
obfuscated bzip2	1369	5682
regexp	289	478
obfuscated regexp	351	1068

Table 2

We pick 50% as the obfuscation level to demonstrate the performance of Turing machine obfuscator while we also conducted experiments with other obfuscation level of 30%, 80% and 100%. Figure 3 demonstrated that with a higher obfuscation level, the number of call graph edge increases which indicates the obfuscated binary become more and more complicated.

5.2 Resilence

A good obfuscation technique should resist deobfuscation tools well. Concolic testing is an advanced deobfuscation techniques aiming at finding bugs or vulnerbilities in software development through symbolic execution. Whereas, it is also used by adversaries to analyse or restore software inner logic eg. control flow graph. KLEE[17] 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 resilence of Turing Machine obfuscator. We used a piece of sample code from KLEE[17] as the subject program. The subject program need to be converted to IR codes since KLEE works on IR level.

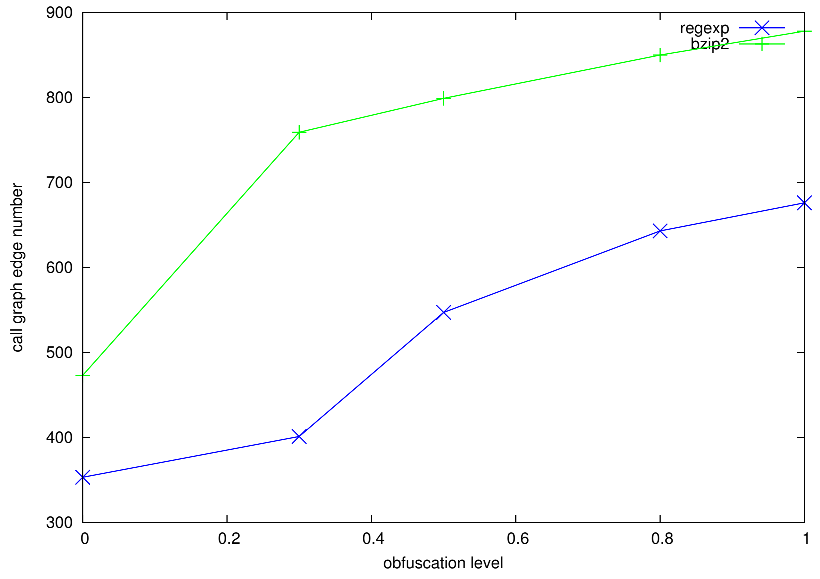


Fig. 7: Call graph Edge Number versus obfuscation level

```
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);
}
```

KLEE could detect three completed 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 contrust, after subjuct program is obfuscated by our Turing machine obfuscator, KLEE could only figure out one completed path. This result proved that Turing machine obfuscator can hinder automated deobfuscation tools from restoring the structure of a subject program.

Due to limited information released by KLEE, we could not dig deeper to understand why KLEE fails. We assume it is because the other constraint solvers can't find another symbolic input to make sure the other branches could also be reached in consequence of the complextiy of 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, authors of [3] calculated the statistics of instruction distribution of both original and obfuscated programs to draw a comparison. If instruction distribution statistics of a obfuscated program is distinct from normal programs(e.g call or jmp instruction proportion is abnormally high), it would be a obvious indicator that denotes this program has been instrumented. We adopted this metric to evaluate our Turing obfuscator. Obfuscation level for stealth evaluation is set to 50%.

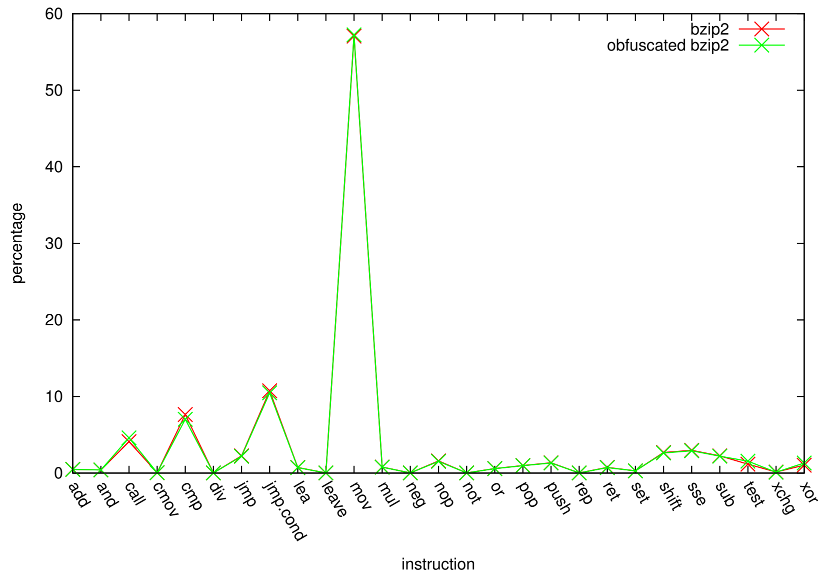


Fig. 8: bzip2 instruction distribution comparison

We classified binary instructions into 27 different classes. Figure 4 and Figure 5 present the instruction distribution of the original and obfuscated version of target programs(bzip2 and regexp). Experiment results indicate that the instruction distribution after obfuscation is very close to the origin distribution. Given the fact that obfuscated binaries contain the same amount of extra instructions from Turing machine obfuscator, slre changed more than bzip2 because the former binary contains only 1391 lines of code and the later binary contains 8117 lines of code. Small instruction distribution variation indicated that our technique could obfuscate programs in stealth.

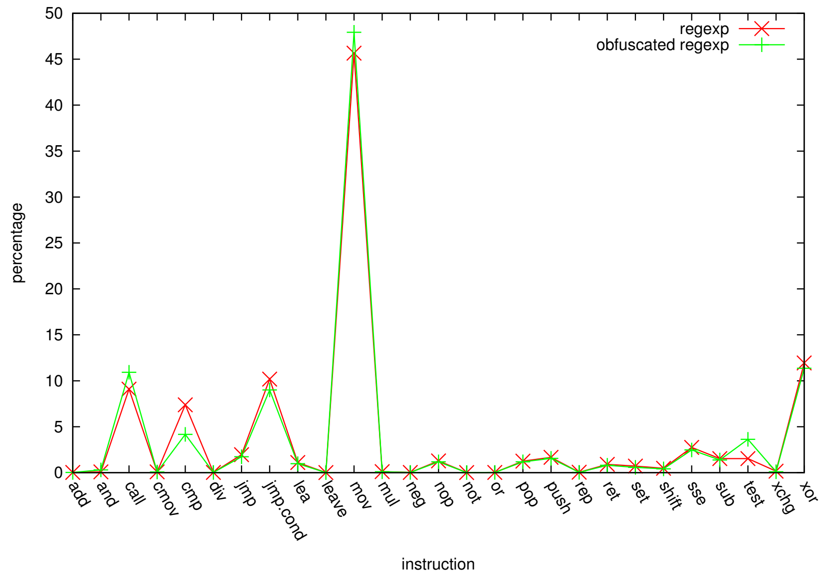


Fig. 9: regexp instruction distribution comparison

5.4 Cost

Software running cost is another important aspect in obfuscation evaluation. In most obfuscation research works, computing overhead increase is inevitable because obfuscation would induce extra instructions execution in one way or another. Measuring time cost is a simple and traditional way to evaluate to what extent an obfuscator induces execution overhead. In evaluation, source programs and obfuscated programs are run on a server with 2 Intel(R) Xeon(R) E5-2690 @ 2.90GHz processors and 125GB system memory. Bzip2 compressed three different sample files and regular expession engine slre rans 149 test cases proviede by [9] in cost evaluation. We ran each program three times and recorded the average time cost as the final result.

Figure 6 demonstrates time cost monotonically increases with higher obfuscation levels for both target binaries. Obfuscated program takes more time to finish running as expected and we believe time cost is still confined to a reasonable level. We also noticed that there is a difference between slopes of the two curves. Turing obfuscator only randomly and statically obfuscates candidates instructions within a binary, so it does not ensure each chosen instruction candidate will be executed in the runtime. The curve slope difference probably result from the uncertainty of obfuscated instruction execution. In addition, the regexp program employs more recursive calls than the bzip2 program, this characteristic may also contribute to the rapid time cost increase of the regexp curve.

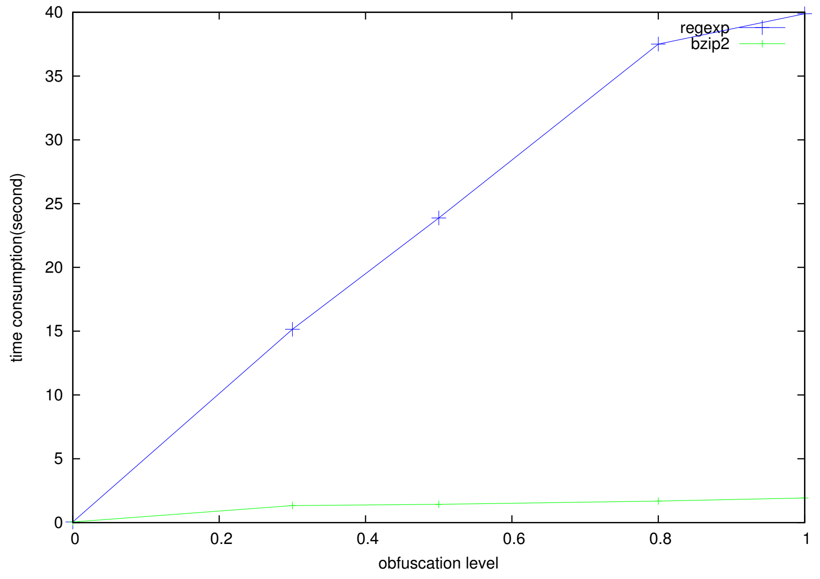


Fig. 10: Time cost versus obfuscation level

6 Discussion

7 Conclusion

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. Ma, Haoyu, et al. "Control flow obfuscation using neural network to fight concolic testing." International Conference on Security and Privacy in Communication Systems. Springer International Publishing, 2014.
3. Wang, Pei, et al. "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. Popov, Igor V., Saumya K. Debray, and Gregory R. Andrews. "Binary Obfuscation Using Signals." Usenix Security. 2007.
7. Wang, Zhi, et al. "Branch obfuscation using code mobility and signal." Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual. IEEE, 2012.
8. `http://www.bzip.org/`
9. `https://github.com/cesanta/slre`

10. Chen, Haibo, et al. ”Control flow obfuscation with information flow tracking.” Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2009.
11. `https://www.hex-rays.com/products/ida/`
12. McCabe, Thomas J. ”A complexity measure.” IEEE Transactions on software Engineering 4 (1976): 308-320.
13. 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.
14. `https://en.wikipedia.org/wiki/Turing_machine`
15. `https://en.wikipedia.org/wiki/Universal_Turing_machine#/media/File:Universal_Turing_machine.svg`
16. `https://en.wikipedia.org/wiki/LLVM`
17. `https://klee.github.io/`