Design and implementation of code obfuscator based on random opcode

Dan Si1,a

1Software College, Sichuan University, Chengdu, Sichuan, 610207, China
aemail: 2018141463142@stu.scu.edu.cn

Abstract. Software obfuscation is a common way of software protection. Source obfuscation and binary obfuscation are unable to achieve universal obfuscation algorithms, while existing compile-time obfuscation algorithms can be effectively restored by symbolic execution. In this paper, an innovative code obfuscation algorithm based on random opcode is proposed. The random number generator is a linear congruence generator written recursively. The algorithm can be used for code obfuscation across languages, platforms and architectures, and has application value for general-purpose software protection. The results of our experiments show this algorithm proposed in this paper can significantly increase the time complexity and space complexity of symbol execution and greatly increase the cost of reverse engineering, which can effectively resist the restore by symbolic execution and has a good confusion effect.

1. Introduction

In recent years, the problem of software protection security related to reverse engineering has been paid more and more attention. Up to now, the problem of protecting the integrity and confidentiality of running code has been solved from two opposite directions: encryption and obfuscation. In the late 1990s, Collberg et al. [1] first proposed the concept and quantitative indicators of code obfuscation in his paper. Chang et al. [2] considered a method based on setting software sentinel guard to achieve tamper-proof, but this method relies too much on sentinels. Horne et al. [3]proposed a testers method, a function similar to a sentinel, which calculates a program's Hash value while a program is running to verify that a program has been modified. An obfuscation scheme for binary code was first proposed by Wroblewski et al. [4]. Linn et al. [5]firstly proposed code disassembler. Popov et al. [6]proposed control flow transfer obfuscations based on exceptions.

Many code obfuscation schemes proposed are designed for specific programming languages or running platforms. There are also many research results of anti-code obfuscation, including semantic analysis and symbolic execution [7]. With the advent of the LLVM compilation architecture, code obfuscation has gradually shifted from language-specific, platform-specific obfuscation to general-purpose obfuscation. Junod et al. [8] combined previous studies and implemented a general code obfuscator based on LLVM compilation architecture, but this method could not effectively resist symbol execution.

In this paper, in view of the defects of Junod et al.’s method, we propose and implement a random opcode based code obfuscation, providing a new idea for anti-symbol execution of code obfuscator. The main contributions of this paper are as follows: (1) design an anti-symbol execution code obfuscation scheme, (2)based on LLVM implement random opcodes based code obfuscation. (3) The random opcode obfuscation algorithm is compared with other compile-time obfuscation algorithms using different quantization indexes.
2. Methdology

The code obfuscation algorithm based on random opcodes is an improvement of control flow flattening which cannot resist symbolic execution. The function's control flowchart would be destroyed, turning the basic blocks within the function into virtual machines with custom opcodes. To describe this process, a new term LLVM command is defined, which stands for LLVM instruction with opcodes and operands. For example, alloca is an LLVM directive, and %1 = alloca i32 is an LLVM command. Each opcode is generated by recursive based random function. Because of the problem of path explosion exists in recursive function call and infinite loop by symbol execution, we realize the resistance of control flattening to symbol execution.

The obfuscation algorithm obfuscates in basic blocks and does not deal with basic blocks containing PHI nodes or exception-handling nodes.

2.1. Random opcode

In the opcode, an opcode is randomly assigned to each LLVM command._OPCODES are allocated in units of LLVM commands, and different opcodes are assigned to the same LLVM instructions. This is due to the limitations of the LLVM IR, which does not allow you to dynamically set instruction operands or virtual registers. Therefore, the control flow chart of the virtual machine will be more complicated. Obfuscated instructions make it harder to reverse the function because the same instruction corresponds to different opcodes. Random opcodes are generated by a recursive linear random number generator. After the execution of the current instruction is completed, the opcode of the current instruction is passed into the random number generation function to generate the opcode of the next instruction. Random number generation is done dynamically in memory, so it is difficult to be analyzed statically. The implementation pseudocode is shown in Figure 1 below.

![Figure 1. Random number generation function pseudocode.](image)

2.2. Virtual machine interpreter

The virtual machine interpreter is a combination of while and switch. Inside the obfuscated function, each obfuscated base block has an interpreter. The interpreter calls the random number generation function to generate the opcodes and selects the corresponding instructions according to the opcodes.

The virtual machine interpreter needs to make sure that the opcodes of the virtual machine are unique in the VM instructions, so first count how many instructions the virtual machine has and then make sure the generated sequence is unique. Meanwhile, because of the loss of the sequence of instructions after virtualization, there will be some virtual registers that are not created but used, so all the initialization instructions need to be forced into the virtual machine initialization instruction block. In the obfuscation algorithm, the basic block containing exception handling, PHI instruction, is skipped. The virtual machine adds three generated basic blocks to the function: virtual machine initialization vmBody, vmBody, and vmDefault. The implementation pseudocode is shown in Figure 2.
2.3. The evaluation index

The quantitative index adopted in this paper is the two quantitative criteria proposed by Zhao Yujie et al. [9] from the perspective of attackers: instruction execution rate and control flow cycle complexity.

2.3.1. Instruction Execution Rate

$I_E$ represents all the instructions generated after disassembly; $I_d$ represents the actual number of instructions executed in execution. The closer $IE$ is to 1, the higher the space efficiency of obfuscation is, and the more memory is saved. The calculation formula is as follows:

$$IE = \frac{I_d}{I_E}$$

2.3.2. Cyclic complexity of Control Flow

The Control Flow Graph of software is transformed into a directed Graph, and the quality of code confusion is calculated with the knowledge and method of Graph theory. Control flow chart is the basis of control flow cycle complexity calculation. In LLVM, the control flow chart is constructed in terms of functions. The calculation formula is as follows:

$$V(G) = e - n + 2$$

$e$ represents the number of edges in the control flowchart, and $n$ represents the number of nodes in the control flowchart. The higher the value of $V(G)$, the better the confusion effect.

3. Experiment& result

3.1. Experimental data

In this paper, we conducted our experiments on data including the C/C++ program "Hello World", the commonly used Hash algorithm MD5, commonly used encryption and decryption algorithm TEA and commonly used encoding algorithm Base64.

The obfuscation algorithm based on random opcode is to obfuscate the control flow chart, so it makes a horizontal comparison with the false control flow and the control flow flattening in OLLVM.

3.2. Experimental steps

Firstly, we use the framework ANGR of symbolic execution to static analyse the control flow chart, and then calculate the control flow cycle complexity. Secondly, Pintools is used to count the RIP address at runtime and IDA static analysis of the disassembly set. The instruction execution rate is
got by dividing by the two. Thirdly, the anti-symbol execution ability of the algorithms are compared, including the duration of symbol execution and the memory required by symbol execution running different confusion algorithms. Lastly, in order to show that the anti-sign execution ability is not the result of simple code stacking, we add the direct running time.

Table 1. Control flow cycle complexity.

| Program name | Obfuscation algorithm | Random opcode | Bogus control flow | Control flow flatten | Not obfuscated |
|--------------|-----------------------|---------------|-------------------|----------------------|----------------|
| Hello World  | 32                    | 8             | 8                 | 8                    | 8              |
| TEA          | 363                   | 36            | 25                | 77                   | 58             |
| MD5          | 874                   | 90            | 140               | 102                  |                |
| BASE64       | 135                   | 156           |                   |                      |                |

Table 2. Instruction execution rate.

| Program name | Obfuscation algorithm | Random opcode | Bogus control flow | Control flow flatten | Not obfuscated |
|--------------|-----------------------|---------------|-------------------|----------------------|----------------|
| Hello World  | 100%                  | 100%          | 100%              | 100%                 | 100%           |
| TEA          | 100%                  | 74.53%        | 100%              | 100%                 | 100%           |
| MD5          | 100%                  | 83.14%        | 100%              | 100%                 | 100%           |
| BASE64       | 100%                  | 84.64%        | 100%              | 100%                 | 100%           |

Table 3. Symbol execution time complexity (s).

| Program name | Obfuscation algorithm | Random opcode | Bogus control flow | Control flow flatten | Not obfuscated |
|--------------|-----------------------|---------------|-------------------|----------------------|----------------|
| Hello World  | 5.50                  | 0.20          | 0.20              | 0.20                 | 0.20           |
| TEA          | 9.36                  | 0.32          | 0.34              | 0.30                 | 180            |
| MD5          | 20.19                 | 0.76          | 1.24              | 180                  |                |
| BASE64       | 168.46                | 3.3           | 5.01              | 206                  |                |

Table 4. Symbol execution space complexity (MB).

| Program name | Obfuscation algorithm | Random opcode | Bogus control flow | Control flow flatten | Not obfuscated |
|--------------|-----------------------|---------------|-------------------|----------------------|----------------|
| Hello World  | 34                    | 24            | 24                | 24                   | 24             |
| TEA          | 274                   | 178           | 250               | 14                   | 14             |
| MD5          | 197                   | 184           | 184               | 180                  | 180            |
| BASE64       | 309                   | 280           | 295               | 206                  | 206            |
### 3.3. Analysis of experimental results

By analyzing the results in Table 1 and Table 2, the random opcode confusion algorithm increases the control flow cycle complexity by 8-10 times as much as other confusion algorithms. The instruction execution rate is 100% except for the false control flow. Because the higher the instruction execution rate, the more space efficient, more space saving and the greater the complexity of control flow cycle, the higher the quality of confusion, the confusion quality of random opcode obfuscation algorithm is better than that of false control flow and control flow flattening in OLLVM.

As shown in Table 3 and Table 5, false control flow and control flow flattening algorithm show a linear correlation between the increase time of symbol execution and the increase time of actual execution, which is basically unable to resist symbol execution. In addition to the exception Base64, the symbol execution time of the random opcode obfuscation algorithm increases by about 28 times compared with the unobfuscated results, while the actual running time only increases by about 1.5 times, indicating that the anti-symbol execution of the algorithm is not the result of code stacking, and the anti-symbol execution of the random opcode obfuscation algorithm is indeed effective. Base64, the exception point, also conforms to this rule as the symbol execution time increases by 60 times, and the actual execution time increases by 10 times. After IDA reverse analysis, switch statement nesting occurred after execution of confuse algorithm due to the existence of switch statement in the original Base64 code, resulting in a sharp increase in jumps, and finally the running efficiency was reduced by 10 times.

Analysis of Table 4 shows that the increase in running memory of random opcode algorithm is not large compared with other algorithms, and the use of obfuscation algorithm will not significantly improve the minimum running configuration, which meets the expected results of code obfuscation. The confounding algorithm has practical value.

### 4. Conclusion

With the development of cloud computing and mobile computing, more and more attention has been paid to the protection and security of software in reverse engineering. It is a very difficult task to protect software from tampering, malicious modification or reverse engineering. One of the most challenging tasks facing information security researchers today is to design encryption schemes and software technologies that protect the integrity and confidentiality of static or running code from white-box attacks. In this paper, we have done 5 comparative experiments on the proposed random opcode obfuscation algorithm. Random opcode obfuscation anti-coincidence execution is really more effective than false control flow and control flow flattening in OLLVM, which is not simply realized by code stacking. After confusion, the time and space complexity of symbol execution significantly increase. In the near future, we will further optimize our algorithms to better protect the software from tampering, malicious modification or reverse engineering so that our algorithms can be widely used in software protection practices.

### Reference

[1] Collberg, C., Thomborson, C., Low, D. (1997) A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, Univ. of Auckland, New Zealand July.

[2] Chang, H. (2001) Protecting software codes by guards. Proc Workshop on Security & Privacy in...
Digital Rights Management. Lecture Notes in Computer Science, 2320:160-175.

[3] Horne, B., Matheson, L., Tarjan, R.E. (2003) Dynamic Self-Checking Techniques for Improved Tamper Resistance. Lecture Notes in Computer Science, 2320:141-159.

[4] Wroblewski, G. (2002) General Method of Program Code Obfuscation. Wroclaw University of Technology, Institute of Engineering Cybernetics.

[5] Linn, C., Debray, S. (2003) Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: 10th ACM(Association for Computing Machinery) Conference on Computer and Communications Security. Washington D.C. pp. 290-299.

[6] Popov, I.V., Debray, S.K., Andrews, G.R. (2008) Abstract Binary Obfuscation Using Signals. In: 16th USENIX Security Symposium. Boston. pp. 275-290.

[7] Xiao, S.T., Zhou, A.M., Liu, L., Jia, P., Liu, L.P. (2018) An Anti-Obfuscation Framework for Virtual Machine Obfuscator Based on Symbol Execution. Journal of Computer Application, 38(06):1745-1750.

[8] Junod, P., Rinaldini, J., Wehrli, J., Michielin, J. (2015) Obfuscator-LLVM -- Software Protection for the Masses. In: IEEE/ACM 1st International Workshop on Software Protection (SPRO). Florence. pp. 3-9.

[9] Zhao, Y.J., Yang, Z.Y., Wang, N., Fang, D.Y., Gu, Y.X. (2012) Evaluation of Code Obfuscating Transformation. Journal of Software, 23(03):700-711.