Bytecode Obfuscation for Smart Contracts

Qifan Yu[1], Pengcheng Zhang[1], Hai Dong[2], Yan Xiao[3], Shunhui Ji[1]
[1]College of Computer and Information, Hohai University, Nanjing, China
[2]School of Computing Technologies, RMIT University, Melbourne, Australia
[3]School of Computing, National University of Singapore, Singapore
1455308407@qq.com; pchzhang@hhu.edu.cn; hai.dong@rmit.edu.au; dcsxan@nus.edu.sg; shunhuiji@hhu.edu.cn

***Abstract*—Ethereum smart contracts face serious security problems, which not only cause huge economic losses, but also destroy the Ethereum credit system. To solve this problem, code obfuscation techniques are applied to smart contracts to improve their complexity and security. However, the current source code obfuscation methods have insufficient anti-decompilation ability. Therefore, we propose a novel bytecode obfuscation approach called BOSC based on four kinds of bytecode obfuscation techniques, which is directed at solidity. The experimental results show that, after the bytecode obfuscation, the failure rate of decompilation tools is over 99% and only a small amount of gas is consumed.**

***Index Terms*—Ethereum, Smart Contract, Bytecode Obfuscation**

I. INTRODUCTION

In Ethereum, smart contracts exist in the form of contract accounts that manage electronic cryptocurrencies stored in the blockchain platform. Currently, Solidity is the first choice for programming smart contracts on the Ethereum platform. Solidity is a high-level language, the compiled bytecode files of which will run on the Ethereum Virtual Machine (EVM).

With the wide application of smart contracts, the security issues of smart contracts have been gradually exposed. For example, The Dao incident resulted in a loss of $60 million [1]. Advances in reverse analysis capabilities have resulted in more decompilation tools. With these tools, it is possible to convert executable bytecode into readable assembly or even source code. People with basic assembly knowledge can use decompilation tools to understand program logic and make unauthorized use, tampering and vulnerability discovery. Source code obfuscation technologies have been applied to smart contracts to improve their security, via enhancing contract complexity and decompilation cost [4]. Since decompilation tools focus on the process of restoring bytecode to source code, source code obfuscation techniques fall short in its ability to invalidate decompilation tools in the bytecode level.

To address the limitation of the existing technique, we propose a novel smart contract bytecode obfuscation tool (named BOSC[1]) that leverages four bytecode obfuscation techniques, namely incomplete instruction obfuscation, false branch obfuscation, instruction rearrange obfuscation and flower instruction obfuscation. By collecting a real smart contract bytecode dataset from Ethereum, we evaluate the anti-decompilation performance of BOSC on three advanced Solidity decompilers. The experimental results show that BOSC can effectively fight against decompilers with low gas consumption.

[1]https://anonymous.4open.science/r/APSEC2022, including datasets.

II. THE BOSC APPROACH

Our approach comprises three steps. After receiving a bytecode file, we perform bytecode cleaning. Next, we execute bytecode obfuscation, which includes incomplete instruction obfuscation, false branch obfuscation, instruction order rearrange obfuscation and flower instruction obfuscation [2]. Finally, we implement bytecode recovery to generate an obfuscated bytecode file. The overall process is shown in Fig. 1.

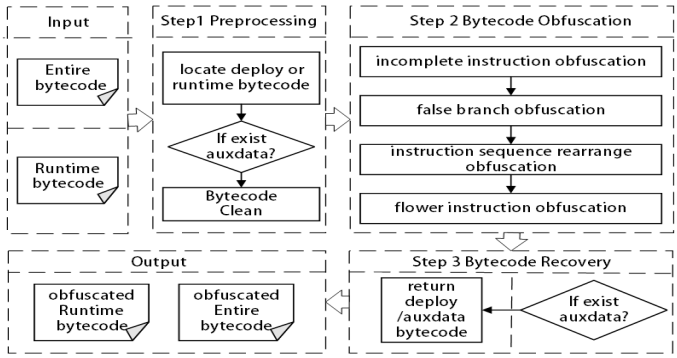

Fig. 1. Flowchart of the bytecode obfuscation

[Step 1] Preprocessing. First, a bytecode file suffixed with .hex is fed into the tool. The bytecode of Solidity contains three parts, which are deployment, runtime and auxdata bytecode. Deployment bytecode is responsible for deployment of a contract. It stores runtime and auxdata bytecode and associate the storage address of both to an contract account. The runtime bytecode is the core expression part of the program logic. Generally decompilers only accept the runtime bytecode for input, so we only perform bytecode obfuscation on the runtime bytecode. The auxdata bytecode is the encrypted fingerprint of the code. This part is used for verification, which will not be executed by the EVM and is optional. In summary, the major purpose of our bytecode cleaning is to extract the runtime bytecode and remove deployment and auxdata bytecode.

[Step 2] Bytecode Obfuscation. We aim to achieve two main goals for bytecode obfuscation. The first is to make decompiling as challenging as possible. The second is to make decompiled code difficult to understand even if the decompilation is successful. The following subsections introduce each bytecode obfuscation method.

[2.1] *Incomplete instruction obfuscation.* The key of the method is to create decompiler errors by inserting an incomplete instruction, which only contains an opcode without

Q. Yu, P. Zhang, H. Dong, Y. Xiao and S. Ji, "Bytecode Obfuscation for Smart Contracts," 2022 29th Asia-Pacific Software Engineering Conference (APSEC), Japan, 2022, pp. 566-567, doi: 10.1109/APSEC57359.2022.00083.

TABLE I
EXPERIMENTAL RESULTS OF ANTI-DECOMPILATION ABILITY.

	Original Bytecodes				Obfuscated Bytecodes			
	PF	FF	SUC	F-Total	PF	FF	SUC	F-Total
OSD	2.5%(5/200)	0.0%(0/200)	97.5%(195/200)	2.5%(5/200)	47.0%(94/200)	52.0%(104/200)	1.0%(2/200)	99.0%(198/200)
Vandal	3.5%(7/200)	0.5%(1/200)	96.0%(192/200)	4.0%(8/200)	89.5%(179/200)	10.5%(21/200)	0.0%(0/200)	100%(200/200)
Gigahorse	1.5%(3/200)	0.5%(1/200)	98.0%(196/200)	2.0%(4/200)	98.5%(197/200)	1.5%(3/200)	0.0%(0/200)	100%(200/200)

an operand or incomplete operand [2]. When a decompiler encounters this instruction fragment, it will naturally read the subsequent data to make it a complete instruction, so that the subsequent instruction cannot be correctly identified.

[2.2] *False branch obfuscation.* Jump instructions are based on calculation results, which rely on the input data. For most jumps, decompilers can only assume each of branches is reachable. Based on this feature, an unconditional jump can be converted into a conditional jump, while in fact one of branches is never reachable and false [2]. Any jumps and loops can be inserted into this false branch to make the decompiler iteratively visit this branch layer by layer until it dies.

[2.3] *Instruction sequence rearrange obfuscation.* It is a simple obfuscation technique, the main mission of which is to change the execution sequence of mutually independent instructions [3]. A pair of independent instructions refer to a precedent instruction and a decedent instruction that do not influence each other. The change of their sequence does not affect the program. However, the sequence reflects developers' design thinking. Hence, changing the order of independent instructions makes the program more difficult to comprehend.

[2.4] *Flower instruction obfuscation.* Flower instructions, also known as dirty bytes, refer to junk instructions or redundant instructions, which are generally meaningless [3]. Inserting flower instructions while ensuring the correct execution of the original program can effectively interfere with the thinking of attacker's decompilation, making it difficult to understand the program content and achieving the effect of obfuscation.

[Step 3] Bytecode Recovery. The previously removed bytecode part, such as deployment and auxdata bytecode is restored in the final generated obfuscated bytecode file.

III. EVALUATION

We evaluate BOSC from three aspects, i.e., consistency, anti-decompilation ability and extra gas consumption. We randomly collect 200 bytecode files of real smart contracts from Ethereum. These bytecode files are relatively complex, with an average bytecode number of $15,725$.

Consistency. The purpose of this experiment is to verify if the original logic of the bytecode does not change. Through a manual review, the logic of these 200 bytecode files has not changed after obfuscation.

Anti-decompilation. Experiments are carried out with three common and advanced Solidity decompilers, i.e., *Online Solidity Decompiler (OSD)*, *Vandal* and *Gigahorse*. The results of the anti-decompilation ability are shown in table I. *PF* (part-failure) refers to that the decompilation tool can output a result, but the result reports an error or lacks the jump logic. *FF* (full-failue) means that the decompilation tool cannot output the result at all. *SUC* (success) refers to the successful output of the decompilation result. *F-Total* is the sum of *PF* and *FF*. The experimental results show that almost all of these decompilation tools can run normally before bytecode obfuscation. However, after the bytecode obfuscation, the overall failure rates of OSD, vandal, and gigahorse respectively reach 99%, 100% and 100%. This result proves that BOSC has an effective anti-decompilation capability.

Extra Gas Consumption. To assess the extra gas consumption, we measure the maximum number of extra instructions introduced for each bytecode obfuscation method to calculate the worst-case gas consumption. We take gwei as 54, which is much larger than the average price. In this case, each unit of gas will cost about 0.0007\$. The average time to run each bytecode obfuscation method is provided, which is acceptable. It can be seen that, in the worst case, BOSC only generates 1180 extra gas consumption, which is worth about 0.82453\$. The experimental results are shown in table II.

TABLE II
EXPERIMENTAL RESULTS OF EXTRA GAS CONSUMPTION

Metrics / Methods	Extra Gas(gwei)	Dollar($)	Time(ms)
Incomplete instruction	200	0.13975	137
False branch	230	0.16071	283
Instruction sequence rearrange	0	0	114
Flower instruction	750	0.52407	159

IV. CONCLUSION

This paper proposes a novel bytecode obfuscation method for Ethereum smart contract, which can greatly improve the anti-decompilation ability of smart contracts and generate only a small amount of extra gas consumption.

ACKNOWLEDGEMENT

This work is funded by the National Natural Science Foundation of China under Grant No.62272145 and No.U21B2016.

REFERENCES

[1] Chen, H. , et al. "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses." ACM Computing Surveys 53.3(2020):1-43.

[2] Wroblewski G . General Method of Program Code Obfuscation. 2002.

[3] Linn C , Debray S . Obfuscation of executable code to improve resistance to static disassembly[C]// Computer and Communications Security. ACM, 2003.

[4] Zhang, M., Zhang, P., Luo, X., & Xiao, F. (2020, December). Source Code Obfuscation for Smart Contracts. In 2020 27th Asia-Pacific Software Engineering Conference (APSEC) (pp. 513-514). IEEE.