EShield: Protect Smart Contracts against Reverse Engineering

Wentian Yan
yanwentian@pku.edu.cn
HCST, CS, Peking University
Beijing, China

Jianbo Gao
Zhenhao Wu
gaojianbo@pku.edu.cn
zhenhaowu@pku.edu.cn
HCST, CS, Peking University
Beijing, China
Boya Blockchain Inc.
Beijing, China

Yue Li
liyue_cs@pku.edu.cn
HCST, CS, Peking University
Beijing, China

Zhi Guan*
guan@pku.edu.cn
National Engineering Research
Center for Software Engineering,
Peking University
Beijing, China

Qingshan Li
liqs@pku.edu.cn
HCST, CS, Peking University
Beijing, China
Boya Blockchain Inc.
Beijing, China

Zhong Chen
zhongchen@pku.edu.cn
HCST, CS, Peking University
Beijing, China

ABSTRACT

Smart contracts are the back-end programs of blockchain-based applications and the execution results are deterministic and publicly visible. Developers are unwilling to release source code of some smart contracts to generate randomness or for security reasons, however, attackers still can use reverse engineering tools to decompile and analyze the code. In this paper, we propose EShield, an automated security enhancement tool for protecting smart contracts against reverse engineering. EShield replaces original instructions of operating jump addresses with anti-patterns to interfere with control flow recovery from bytecode. We have implemented four methods in EShield and conducted an experiment on over 20k smart contracts. The evaluation results show that all the protected smart contracts are resistant to three different reverse engineering tools with little extra gas cost.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; *Software security engineering*; • **Theory of computation** → *Program analysis.*

KEYWORDS

Reverse Engineering, Smart Contract, Blockchain, Ethereum, Program Analysis

*Corresponding Author

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3404365

ACM Reference Format:
Wentian Yan, Jianbo Gao, Zhenhao Wu, Yue Li, Zhi Guan, Qingshan Li, and Zhong Chen. 2020. EShield: Protect Smart Contracts against Reverse Engineering. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404365

1 INTRODUCTION

Smart contracts are programs running on blockchain and can be invoked through transactions. Due to the irreversible nature of transactions, blockchain-based applications have been widely adopted in recent years, covering various scenarios including games, gambling, finance, etc [5]. All the smart contracts deployed on blockchain are publicly visible and the execution of transactions are deterministic, which makes it difficult to generate random numbers in Ethereum for game and gambling applications [1, 11]. Therefore some applications have to generate pseudo-random numbers through complex operations and make the smart contracts with key random functions closed source. However, the contracts can be analyzed using reverse engineering tools (*e.g.* Erays [13]) and players can utilize the information to increase the chance of breeding profitable kitties, which may destroy randomness of the game and lead to a reduction in the value of rare kitties. Except for generating randomness, developers also make the contracts closed source for security reasons to prevent attackers from analyzing, but the contracts still can be decompiled and exploited by analyzing tools [7, 10].

Although anti-reverse engineering has become a strong need of developers, there are few effective tools for smart contracts. Due to the special architecture of Ethereum, the existing anti-reverse engineering techniques on other platforms such as anti-debugging and obfuscation [4, 8] are either ineffective or too expensive for Ethereum smart contracts. The main challenges of protecting smart contracts against reverse engineering are as follows.

Challenge 1. Non Von Neumann Virtual Machine. The Ethereum virtual machine does not follow the standard Von Neumann architecture and the code of smart contracts is stored in

a virtual ROM rather than generally-accessible memory or storage [11]. Therefore the code cannot be modified during execution, which makes it impossible to execute encrypted code in Ethereum.

Challenge 2. Gas Cost. Gas is the fuel of computation in Ethereum and users are charged according to the executed bytecode. Obfuscation requires adding a large amount of extra instructions into bytecode and leads to a significant increase in gas cost, which makes it unacceptable to developers and users.

EShield Solution. To address the aforementioned challenges, we have developed EShield for protecting smart contracts against reverse engineering. The key insight behind EShield is to increase the difficulty of recovering control flow graph (CFG) by interfering with identifying the connections between basic blocks.

EShield first builds the CFG for smart contracts, then modifies the bytecode and inserts four anti-patterns into basic blocks to ensure the modified bytecode is equivalent to the original bytecode, finally generates the protected bytecode of smart contracts. We conducted experiments on over 20K smart contracts and all the protected bytecode cannot be decompiled by existing reverse engineering tools.

2 OVERVIEW

The general work flow of EShield is shown in Figure 1. In detail that EShield consists of four components. It takes runtime bytecodes of Ethereum smart contract as input and put them into an analyzer. The analyzer recovers the CFG and detects the suitable position to insert the anti-patterns. The information will be transmitted to the original constructor where the original bytecode will be reconstructed to be suitable to insert the anti-patterns. At the same time, the pattern constructor produces the anti-patterns with the obfuscated jump address. In the end, the bytecode constructor will insert the anti-patterns into the reconstructed original bytecodes and generates the pattern bytecodes.

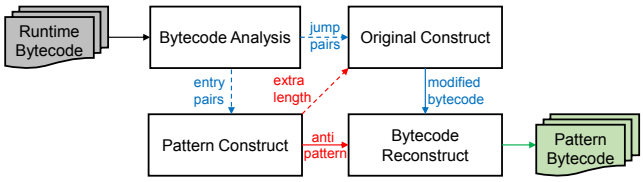

Figure 1: The framework of EShield

2.1 Bytecode Analysis

EShield first uses the evm cfg-builder to recovery the CFG of smart contract bytecodes. It marks every `JUMP` or `JUMPI` instruction with their jump addresses and keeps them in a key-values pairs which named as jump-pairs. The analyzer will check all the jump-pairs and add other three values in each pair. First value is the function address of `JUMP(I)`, second is the position between `JUMP(I)` and its jump address(the address of `JUMP(I)` is ahead of or behind the jump address). Third is the length of jump address. These three values are important in the step of original construct.

The control flow of smart contract can be split into blocks and each block represents a function. The entry can be recognized by an sequence of instructions such as `EQ-PUSH-JUMPI`. The entry detector will analyze all the instructions of entry and mark `JUMPI` with its function address, as we called entry-pairs. Every entry can be used to insert the anti-patterns to generate the security enhanced bytecode.

2.2 Anti-pattern Construct

Pattern 1. The operations in memory cannot be handled well in current reverse tools, so that it can be utilized to obfuscate the jump address. EShield use `MSTORE` to write the jump address into memory and then use `MLOAD` to load it into stack. The instruction `JUMP[I]` will load the function address and jump to the entry of the target function as normal. The instructions can be executed accurately but the reverse tools cannot analyze the real jump address.

Pattern 2. Pattern 2 is similar with pattern 1. It utilizes the storage to handle the jump address. The `SSTORE` instruction stores the function address into storage and `SLOAD` loads it into stack, which can be used to execute `JUMP[I]` accurately.

Pattern 3. EShield uses `SHA3` and a series of operations to get the function address. `SHA3` is also a troublesome instruction which the current reverse tools can not handle. They usually generate a formula to represent the execution result of `SHA3` and won't calculate the value. EShield uses a clever but simple method to generate function address by `SHA3`. EShield uses `SHA3` to generate two identical hash strings and divide the two value to get 1. Then multiplies the address and 1 to get the final address. Inspired by this idea we can use more complex methods to generate the function address.

Pattern 4. Using the call value of external smart contract to produce the jump address is also an effective method to fight against the reverse analysis. We prepared a specific external contract which can return a fixed value by a `CALL`. Then EShield utilizes the fixed value to produce the jump address just like the way in pattern 3.

All the patterns above will increase the length of the bytecode, so the pattern constructor will transmit the number of extra length to original constructor.

2.3 Original Construct

With the extra length and jump-pairs, original constructor will reconstruct the original bytecode to make it suitable for the anti-patterns. There are three steps to finish the entire process of reconstruction. Step 1, if the jump address is behind the position of entry, the constructor will add extra length and the address value. Step 2, calculate all the extra length produced by increasing length of jump address. Maybe the length of original jump address is only 1 byte, then after adding the extra length of pattern it comes to 2 bytes.This situation will probably lead to an increase in it's own length of address. The last step, calculate the change of length in other addresses which result from the situation in step 2.

3 USING ESHIELD

As illustrated in Figure 2, EShield can be divided into four parts in implementation, bytecode analyzer, pattern constructor, original constructor and bytecode constructor. Batches of Ethereum runtime bytecodes can be saved in files and passed into EShield as input.

User can select which pattern will be used to protect smart contract. In the end, EShield will generate a single pattern bytecode file for each pattern the user selected.

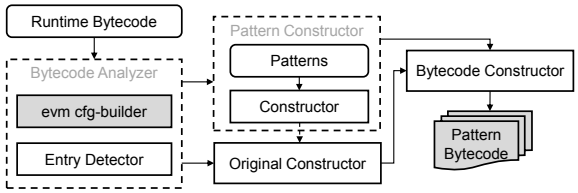

Figure 2: The architecture of EShield in implementation

Bytecode Analyzer. Bytecode analyzer consists of two components, evm-cfg-builder[1] and entry detector. The evm-cfg-builder is a reliable foundation for building program analysis tools in EVM. It extracts the CFG of bytecode so that EShield can mark all the `JUMP(I)` with jump addresses and other necessary values to generate the jump-pairs, then transmits them to original constructor. Entry detector will recognize and mark every entry of function, then transmit them to the pattern constructor.

Pattern Constructor. Pattern contructor receives the entry pairs from bytecode analyzer and generates the anti-patterns for each pattern. The jump address in every anti-pattern will be modified to be suitable for inserting. The anti-patterns will be transmitted to the bytecode constructor and the extra length of anti-pattern will be passed into original constructor.

Original Constructor. After receiving the original bytecode and extra length, original constructor begins to reconstruct the bytecode. It will consider all the situations that can make the jump address change and modify them correctly. The modified bytecode will be tansmitted to the final constructor to produce the protected bytecodes.

Bytecode Constructor. Bytecode constructor is the final constructor to generate the pattern bytecodes. It will insert the anti-patterns into the modified bytecode to replace the old entry bytecode. EShield will generate a single file for each chosen pattern.

EShield has been deployed as a web service and the command-line version with the same functions is also available. Figure 3 shows the web page of EShield. Users can input the bytecodes or the bytecode file path. After selecting the patterns and setting the output path, users can generate the pattern bytecodes or files by clicking the GO button. At last the pattern bytecodes will be displayed in the page, or saved into files.

4 PRELIMINARY EVALUATION

We downloaded the runtime bytecode of 20,266 smart contracts deployed on Ethereum mainnet and used EShield to generate four different patterns of protected bytecodes for each smart contract. The experiments were conducted on a virtual machine with 1 vCPU, 6G RAM and Ubuntu 18.04 as the OS.

To evaluate the effectiveness of EShield, we chose three reverse tools, **Erays** [13], **Vandal** [2, 7] and **Gigahorse** [6] as benchmarks.

[1] https://github.com/crytic/evm_cfg_builder/

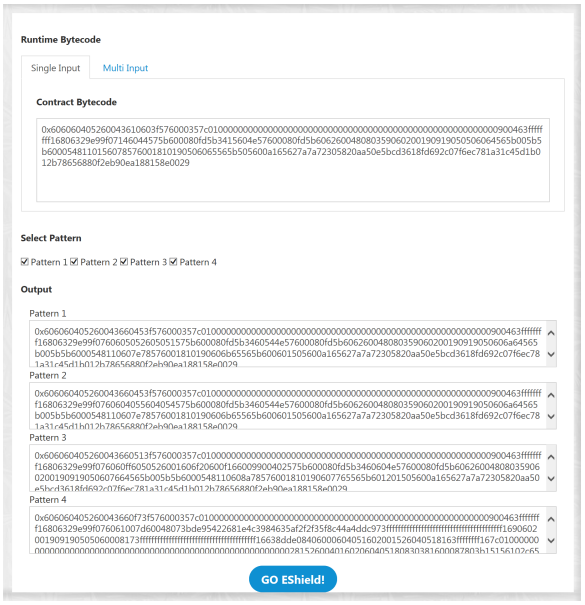

Figure 3: EShield Screenshot

These reverse tools take runtime bytecode as input and recover CFG and generate other forms of high-level representations, which are the state of art reverse analysis tools of smart contracts.

In addition, we use the following criteria to evaluate the result of reversing analysis, *i.e.* Partially Failed (PF) and Entirely Failed (EF). If a reverse tool generates a wrong result, we consider this as a PF case and the rate of PF cases in the dataset is PFR; if the reverse tool cannot generate any results, it is called a EF case and the rate is EFR.

The way to determine the correctness depends on the results generated by the three reverse tools. **Erays** converts runtime bytecode to a CFG in PDF format. If Erays cannot analyze the jump address, it will miss a function block and create fewer PDF files than original one, this situation is a PF case. If there is no PDF file, it is an EF case. **Vandal** converts runtime bytecode into semantic logic relations and CFG. If Vandal cannot find the jump address, it will miss a block and report an error message, which is called a PF case. If there is no result, it is called an EF case. **Gigahorse** is the same as Vandal.

As shown in Table 1, all the four patterns can effectively fight against the reverse tools. The total failed rate (PFR + EFR) of each pattern achieves 100%, no matter which reverse tool was selected. The results show the effectiveness of EShield.

Since different reverse tools have different forms of representations to record the control flow, they have different ways to deal with the failure of CFG recovery. Some reverse tools will terminate directly when there is an error, but others may not. That's why the PFRs and EFRs are so different between different reverse tools.

For each reverse tool, the experiment results are similar even though the patterns are different. It strongly proved that all the four anti-patterns are effective and EShield is enable to fight against reverse analysis.

Table 1: Evaluation Results. P: Patterns

P	Tools	PF cases (PFR)	EF cases (EFR)	Total
1	Erays	272 (1.34%)	19994 (98.66%)	**20266 (100%)**
	Vandal	19508 (96.26%)	758 (3.74%)	**20266 (100%)**
	Gigahorse	20266 (100%)	0 (0%)	**20266 (100%)**
	Average	**13349 (65.87%)**	**6917 (34.13%)**	**20266 (100%)**
2	Erays	272 (1.34%)	19994 (98.66%)	**20266 (100%)**
	Vandal	19515 (96.29%)	751 (3.71%)	**20266 (100%)**
	Gigahorse	20266 (100%)	0 (0%)	**20266 (100%)**
	Average	**13351 (65.88%)**	**6915 (34.12%)**	**20266 (100%)**
3	Erays	272 (1.34%)	19994 (98.66%)	**20266 (100%)**
	Vandal	19512 (96.28%)	754 (3.72%)	**20266 (100%)**
	Gigahorse	20266 (100%)	0 (0%)	**20266 (100%)**
	Average	**13350 (65.87%)**	**6916 (34.13%)**	**20266 (100%)**
4	Erays	272 (1.34%)	19994 (98.66%)	**20266 (100%)**
	Vandal	19517 (96.30%)	749 (3.70%)	**20266 (100%)**
	Gigahorse	20266 (100%)	0 (0%)	**20266 (100%)**
	Average	**13351 (65.88%)**	**6915 (34.12%)**	**20266 (100%)**

Table 2: Cost of Using EShield. P: Patterns, $Extra_{Create}$: Extra gas cost of Creating protected contracts, $Extra_{Dollars}$: Extra cost in dollars, $Extra_{Call}$: Extra gas cost of Calling protected contracts, $Extra_{Dollars}$: Extra cost in dollars, Time: average Time cost of running EShield.

P	$Extra_{Create}$	$Extra_{Dollars}$	$Extra_{Call}$	$Extra_{Dollars}$	Time
1	1200	0.0084	21	0.0001	1.20s
2	1200	0.0084	20212	0.1418	1.19s
3	3600	0.0253	82	0.0006	1.22s
4	37636	0.2640	33297	0.2336	1.32s

Table 2 illustrated the average extra consumption of gas in each pattern. We can exchange the gas to dollar by the time of 9th April in 2020. As we use the default gas price(1 gas unit=41 Gwei, which is much higher than usual) to calculate that 1000 gas worth $0.00701551. That means when user creates or calls a protected smart contract, in the worst case it will cost only $0.2640 and $0.2336 respectively. In the best case that creating a smart contract protected by pattern 1 and 2 will only cost about $0.0084, and calling a contract protected by pattern 1 and 3 will only cost about $0.0001 and $0.0006 respectively. The extra cost of using EShield is very low and acceptable to both developers and users. Table 2 also shows the average time EShield spends in generating protected contracts. The average time it costs was between 1.19s and 1.32s. Considering the average size of original bytecode we used is 6.46KB, these results show that EShield is efficient enough to be used in practice.

5 RELATED WORK

Research on reverse engineering and anti-reversing has been in progress for decades and many approaches focusing on different platforms were proposed. As mentioned above, there are some papers about decompiling and analyzing EVM bytecode as well, however, to the best of our knowledge, there is no previous work on protecting smart contracts against reverse engineering.

Porosity [9] is the first reverse engineering tool for Ethereum smart contracts, which can generate readable Solidity-like source code from EVM bytecode. It is removed from benchmarks in our experiments due to the high failure rate. Madmax [7] is a static program analysis tool for detecting gas-focused vulnerabilities in smart contracts and it uses Vandal [2] to decompile bytecode. Gigahorse [6] is an alternative to Vandal and performs better in decompiling unmodified bytecode.

There are some anti-reversing techniques available on other platforms. Software on Windows platform often use anti-virtualization and anti-debugging to prevent analyzing [3]. Android developers also use packers to encrypt bytecode and decrypt it to memory at runtime for protection [12]. However, these approaches cannot be applied to the smart contract due to the different architectures.

6 CONCLUSION

In this demo paper, we presented EShield, a security enhancement tool for protecting Ethereum smart contracts against reverse engineering. EShield performs bytecode replacement with four different patterns to interfere with control flow graph recovery. EShield managed to protect all the smart contracts from three reverse engineering tools with little extra cost in the evaluation. In the future, we plan to explore more methods and generalize the technique to other blockchain platforms.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under the grant No. 61672060.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*. Springer, 164–186.
[2] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).
[3] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 177–186.
[4] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. 2007. Software protection through anti-debugging. *IEEE Security & Privacy* 5, 3 (2007), 82–84.
[5] Jianbo Gao, Han Liu, Yue Li, Chao Liu, Zhiqiang Yang, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. Towards automated testing of blockchain-based decentralized applications. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 294–299.
[6] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1176–1186.
[7] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116.
[8] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.
[9] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25 (2017), 11.
[10] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.
[11] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[12] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 358–369.
[13] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1371–1385.