EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine

Fuchen Ma*, Ying Fu†, Meng Ren‡, Mingzhe Wang†, Yu Jiang†, Kaixiang Zhang§, Huizhong Li§, and Xiang Shi§
* Beijing University of Posts and Telecommunications, China
† Tsinghua University, China
‡ Sun Yat-sen University, China § WeBank, China
mfc@bupt.edu.cn,fy17@mails.tsinghua.edu.cn, renm8@mail2.sysu.edu.cn, wmzhere@gmail.com,
jiangyu198964@126.com, {kxzhang, wheatli, jimmyshi}@webank.com,

***Abstract*—Attacks on transactions of Ethereum could be dangerous because they could lead to a big loss of money. There are many tools detecting vulnerabilities in smart contracts trying to avoid potential attacks. However, we found that there are still many missed vulnerabilities in contracts. Motivated by this, we propose a methodology to reinforce EVM to stop dangerous transactions in real time even when the smart contract contains vulnerabilities. Basically, the methodology consists of three steps: monitoring strategy definition, opcode-structure maintenance and EVM instrumentation. Monitoring strategy definition refers to the specific rule to test whether there is a dangerous operation during transaction execution. Opcode-structure maintenance is to maintain a structure to store the rule related opcodes and analyze it before an operation execution. EVM instrumentation inserts the monitoring strategy, interrupting mechanism and the opcode-structure operations in EVM source code. For evaluation, we implement EVM* on js-evm, a widely-used EVM platform written in javascript. We collect 10 contracts online with known bugs and use each contract to execute a dangerous transaction, all of them have been interrupted by our reinforced EVM*, while the original EVM permits all attack transactions. For the time overhead, the reinforced EVM* is slower than the original one by 20-30%, which is tolerable for the financial critical applications.**

***Index Terms*—Blockchain security, Ethereum, EVM defending**

I. Introduction

Ethereum is a platform for developers to write their own applications using blockchain techniques. Attacks on the programs deployed on Ethereum platform could cause a big loss of money. In April 2018, a hacker attacked BEC exploiting a data overflow vulnerability in Ethereum ERC-20 contract and successfully transferred innumerable BEC tokens to two other addresses. As a result, a large number of BECs in the market were sold, and the value of the digital currency almost reached zero, which brought a devastating blow to BEC market transactions. Currently, the communities are paying increasingly more attention to the security of Ethereum platform.

To solve the security problems, many researchers attempted to improve the robustness of smart contracts. Testing tools for smart contracts mainly leveraged the techniques of fuzzing and symbolic execution. Fuzzing based tools include Echidna [5] and ContractFuzzer [6], [9]. These tools simulate the execution of a large number of transactions to find vulnerabilities in contracts. Some well-known symbolic execution tools are Oyente [7], [8], Manticore [3] and Mythril [2]. Symbolic execution tools could trigger critical paths and detect security vulnerabilities. All of these tools perform well in detecting potential threats in smart contracts.

However, in the industry practice, we find that the above tools tend to miss certain vulnerabilities. For example, we run ContractFuzzer on a contract with a known timestamp bug for 2 hours, but the bug was not detected. This indicates that it is incomplete to focus only on contract level, so we try to solve this problem at EVM level. In this paper, we propose a methodology to reinforce the EVM platform. It consists of three steps: monitoring strategy definition, opcode-structure maintenance and EVM instrumentation. Monitoring strategy definition provides a specific way to decide whether there is a dangerous operation during the execution of transactions. Opcode-structure maintenance is to maintain a structure to store the interesting opcodes for analysis before executing any operation. EVM instrumentation is the process to insert the monitoring strategy, interrupting mechanism and the opcode-structure operations in the proper place of the EVM source code. In this way, the reinforced EVM* could stop dangerous transactions in real time.

The proposed methodology is scalable to be implemented on different EVM platforms and bug types. For evaluation, we implement our methodology on js-evm, and two monitoring strategies are defined in our implementation to prevent overflow and timestamp bugs. For the instrumentation part, we instrument the monitoring strategies and throw an exception when encountered an unsafe action. A stack is used to store the rule related opcodes as well as the operands if any. 10 smart contracts are collected online with known overflow and timestamp bugs. Then we made a dangerous transaction on each contract on the original EVM and the reinforced EVM*. **None of the dangerous transactions are stopped by the original EVM, while all the transactions on the reinforced EVM* have been interrupted. For the time overhead, the EVM* with the overflow strategy is slower than the original EVM by 22.16%, the EVM* with the timestamp strategy is slower by 28.98% and the EVM* with both strategies is slower by 32.96%.** For the financial critical

applications, the overhead is tolerable to ensure the security. Furthermore, when we apply some open-source fuzz testing tool such as ContractFuzzer to test those vulnerable contracts, many bugs would be missed with two or more hours testing.

The contributions of our work lay on the following aspects:

1) We proposed a framework of reinforcing EVM to prevent dangerous transactions at real time, which is scalable for different EVM platforms such as py-evm and js-evm, and different bug types such as overflow and reentry bugs.
2) We implemented a reinforced EVM on js-evm for overflow and timestamp vulnerabilities. We evaluate the effectiveness of the original EVM and the reinforced EVM*. The reinforced EVM* could successfully stop dangerous transactions from execution.

II. Related Work

We discuss the most related work aiming at the detection of the vulnerabilities in smart contracts and the semantics formalization of EVM.

Smart contract Testing: Fuzzing has been widely used in traditional software vulnerability detection [13]–[16] and fuzzing based tools working on the smart contract simulate plenty of transactions on a private chain. Each transaction contains all the functions of a specific smart contract, but these functions are fed into the transaction in different orders with different inputs. Among these tools, the way to define bugs is different. Echidna [5] takes a list of invariants (properties that should always remain true) as input. For each invariant, it generates random sequences of calls to the contract and checks if the invariant holds. If there is a way to falsify the invariant, Echidna will report it as a bug. Whereas, ContractFuzzer [6], [9] defines test oracles to detect security vulnerabilities.

Symbolic execution tools could auto-generate inputs to trigger different unique code paths, and trace the inputs which crashed the program. Manticore [3] could trigger a key path that may cause a crash and expose its analysis engine via Python APIs. Moreover, Manticore not only aims at symbolic testing for solidity scripts, but is also able to analyze Linux ELF binaries(x86, x86_64 and ARMv7). Mythril [2] combines concolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities. Oyente [7], [8] is a symbolic execution tool that works directly on EVM byte code without access to the high level representations.

EVM semantics formalization: Semantics formalization on EVM tries to tackle a complex mix between requirements for high assurance and a rich adversarial model of Ethereum. KEVM [11] is the first fully executable formal semantics of the EVM, created based on a framework for executable semantics, the K framework. A Lem implementation of EVM [12] provides a formal specification of the interface between a smart contract execution and the rest of the world. Lem is a language designed to compile to various interactive theorem provers, including Coq, Isabelle/HOL, and HOL4. This EVM defination can be used to prove invariants and safety properties of Ethereum smart contracts.

Main difference: Unlike the works mentioned above, we do not provide a detecting tool to discover vulnerabilities in smart contact or formalize the EVM for verification. We propose a method to reinforce the EVM. The reinforced EVM* could make up for the detecting tools. The vulnerabilities missed by those tools could be stopped if it occurs in a transaction.

III. Reinforcement Methodology

We now describe the methodology to reinforce the EVM. The framework of the reinforced EVM* contains three main components as Figure 1 shows. Monitoring strategy refers to the method to define whether an opcode sequence is dangerous and how to stop a dangerous transaction. Opcode-structure maintenance contains three parts – initialization, record and analysis for run-time decision. The last component is EVM instrumentation. In this component, monitoring strategies and structure operations will be inserted into the EVM source code, resulting in the reinforcement EVM*.

A. Monitoring Strategy

Monitoring strategy is the most important part of the reinforced EVM*. The specific strategy consists of two parts: interesting opcodes and the constraints for the opcode sequences. As an example, we will explain how to define strategies of overflow bugs and timestamp bugs.

1) Overflow Bugs: To infer if an overflow action occurs in a transaction, the interesting opcodes may contains ADD, SUB, MUL, ADDMOD, MULMOD and EXP. In an overflow situation, the constraints needed to be satisfied are listed as below. The violation of any of these constraints could possibly lead to an overflow problem.

- If two positive numbers perform an adding operation, but the result of the operation is negative
- If two negative numbers perform an adding operation, but the result of the operation is positive.
- If a positive number subtracts a negative number, but the result is negative.
- If a negative number subtracts a positive number, but the result is positive.
- If two positive numbers perform an multiplication operation, but the result of the operation is negative.
- If two negative numbers perform an multiplication operation, but the result of the operation is negative.

2) Timestamp Bugs: The sensitive opcode that timestamp bugs concerned about is TIMESTAMP. If the value put in call() function, which is the beginning of a transaction, is bigger than zero, or the call() function tries to send some ether to other contracts, there possibly is a timestamp bug if the TIMESTAMP opcode occurs.

B. Opcode-Structure Maintenance

Opcode-structure is a user-defined structure to record interesting opcodes and run-time information related to the analysis of dangerous actions. In each call process, the structure will first be initialized. Before the execution of each opcode, the structure will be updated and analyzed. If the opcode

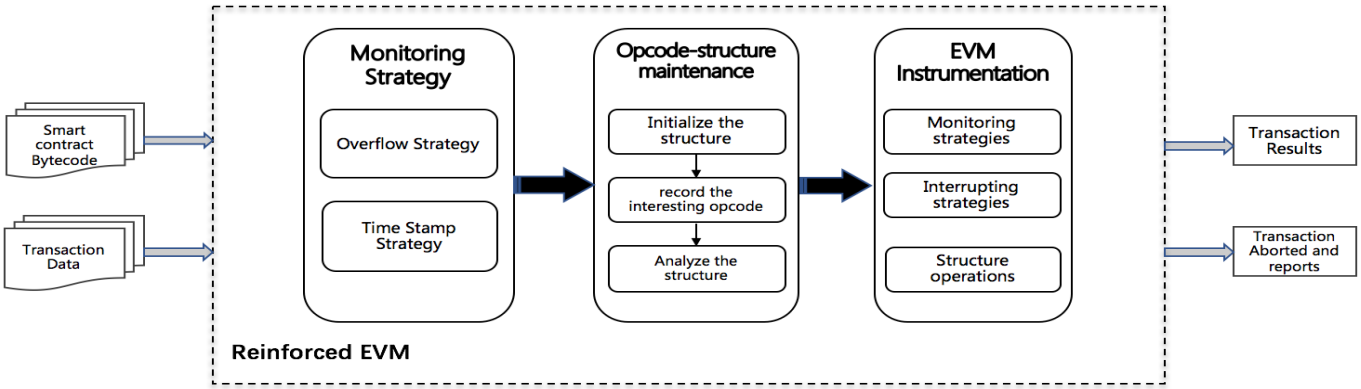

Fig. 1. The framework of the reinforced EVM*. It mainly consists of three components: Monitoring strategy definition for run-time bug detection, Opcode-structure maintenance for operation monitoring, and EVM instrumentation for transaction decision and protection.

sequences in the structure are regarded to be dangerous that violate any constraint defined in the module of monitoring strategy, the interrupt procedure will be executed.

The candidate structures for the implementation of opcode-structure could be: stack, queue, tree and some other structures. The choice of the structure should depend on the monitoring strategy. If the strategy concerns about only the adjacent relationship among opcodes, a stack or a queue may be a better choice. If the strategy focuses on more complex relationships among opcodes, a tree structure may be considered. In our implementation, we use stack as the opcode-structure.

C. EVM Instrumentation

EVM instrumentation module instruments monitoring strategies and interrupting mechanism in the EVM source code. Monitoring strategies should be instrumented in a new file in the EVM project folder. Each strategy needs to be wrapped into a function that returns a boolean type. The interrupting mechanism could be wrapped into a function which is able to terminate the EVM. As for the operations of opcode-structure, the initialization process of the structure should be inserted in the place after where the CALL opcode is detected. Right before the execution of each operation, the opcode-structure should be tested using each monitoring strategy function.

The overall instrumentation process is shown in Algorithm 1. The input of the instrumentation process is the monitoring strategies and interrupt mechanism. As line 1-2 presents, new functions should be defined to implement the monitoring and interrupting process. For each opcode of the transactions, if it is a CALL opcode, a new stack named op_Stack is initialized and the first item is pushed into the stack as shown in line 4-7. Whereas, if it is an interesting opcode we concerned, it will first be pushed in the stack. Then each monitoring strategy will test whether the current stack is safe. If the stack is detected to be dangerous by any strategy, the execution is interrupted, which is shown in line 10-13. If no dangerous actions are detected, the opcode will be executed as line 15 shows.

Algorithm 1: Instrumentation process

```
   Input : monitoring_strategies and Interrupt
           mechanism
1  define_fuc(monitoring_strategies);
2  define_fuc(Interrupt);
3  foreach opcode op of the transaction do
4      if op.isCall() then
5          // if opcode op is the first opcode of a Call
           operation
6          op_Stack = new OP_STACK();
7          op_Stack.push(new Item(CALL,
             op.operands()));
8      else if op.isInteresting() then
9          op_Stack.push(new Item(op.name(),
             op.operands()));
10         foreach monitoring_strategies st do
11             if !Test(st, op_Stack)) then
12                 Interrupt();
13         end
14     end
15     execute(op);
16 end
   Output: Reinforced EVM
```

IV. Evaluation

In our evaluation of the reinforced EVM*, we will answer the following questions:

Q1. Can the reinforced EVM* platform detect vulnerabilities in transactions and stop the vulnerable executions?

Q2. What is the time overhead of the reinforced EVM* on executing different transactions?

A. Data and Environment Setup

To evaluate the effectiveness of the reinforced EVM*, we selected 10 real-world contracts with known overflow and timestamp bugs. The contracts are compiled into abi (Application Binary Interface) files and binary files. Then we used each

contract to execute a dangerous transaction both on the original EVM and the reinforced EVM* to find out whether they could stop the dangerous transactions from execution. Furthermore, we developed and initialized a simple fuzzing environment based on ContractFuzzer to imitate the transaction process. It takes an abi file of the contract as input. In a transaction, each function will be executed random times with arbitrary parameters. The environment augmented with the ability to detect overflow bug and timestamp bug, will create lots of transactions, to see whether these bugs could be detected, in the duration time of 2 hours in this experiment. The version of the solc compiler we used is 0.4.25, the operating system is Linux x86_64.

To answer the second question, we use the fuzzing environment to make transactions on the original EVM and reinforced EVM*. We calculate the amount of transactions made in the time limit on both versions to evaluate the time overhead.

B. Effectiveness of the Reinforced EVM

Table I presents the experimental results of the reinforced EVM* compared with the original EVM. The circle symbol in the table means all the transactions based on the contract executed till the end, while the cross symbol in the table means that the transactions are stopped.

TABLE I
THE EFFECTIVENESS OF REINFORCED EVM* W.R.T. ORIGINAL EVM

Contract Number	Fuzzing tool	Original EVM	Reinforced EVM*	Bug type
Contract_1	detect	○	×	Overflow
Contract_2	detect	○	×	TimeStamp
Contract_3	not detect	○	×	Overflow
Contract_4	not detect	○	×	Overflow
Contract_5	not detect	○	×	Overflow
Contract_6	detect	○	×	TimeStamp
Contract_7	detect	○	×	TimeStamp
Contract_8	detect	○	×	Overflow
Contract_9	not detect	○	×	Overflow
Contract_10	not detect	○	×	Overflow

From the third and the fourth columns of the table, we found that for the original EVM, all the dangerous transactions have been permitted to be running. However, all the dangerous transactions have been stopped from executing by the reinforced EVM*. Furthermore, as the second column shows, only 50% of the vulnerabilities have been detected by the fuzzing environment, which means the detected vulnerabilities could be avoided with manually revision of contracts, and the other 50% would remain exploitable. The reinforced EVM* could successfully stop the dangerous transactions from executing when any problems detected, which could effectively ensure the security of the transaction, even when the deployed contract has vulnerabilities.

Among the five contracts not detected by fuzzing, we analyzed one of them named overflow_simple_add.sol. The source code of the contract is:

```
1  pragma solidity 0.4.24;
2  contract Overflow_Add {
3      uint public balance = 1;
4      function add(uint256 deposit) public {
5          balance += deposit;
6      }
7  }
```

This contract has only one function named add, and this function receives an input named deposit whose type is uint256. In the function add, a variable named balance is added with the input variable and the result will be stored in the variable balance. However, because the variable balance's value is 1, only if deposit's value is the maximum number that a uint256 could represent, the overflow would occur. Fuzzing tools are hard to generate an input which happen to be the maximum number of uint256. So the overflow situation could hardly occur. But when we input the maximum value to make a dangerous transaction, the reinforced EVM* successfully stopped the transaction from executing. It is a common problem among fuzzing tools and symbolic execution tools that testing could not find some bugs that will occur in some extreme cases. However, reinforced EVM* is able to detect the bug in real time and stop the transaction timely.

C. Time Overhead of the Reinforced EVM*

We count the total transactions made during the experiment of each contract. The results are shown in Figure 2.

The unit of the number in the table is counted by transactions per second, which is abbreviated as tx/s. We calculate the transactions made with four types of EVM: the original EVM, the EVM* with overflow strategy, the EVM* with timestamp strategy and the EVM* with both strategies. For each version, we calculate the minimum, maximum and the average amount of transactions made on each contract. As the data shows, the amount of transactions are greatly affected by the size of the contract because the larger the contract is, the more functions a transaction needs to call. Besides, the reinforced EVM* with overflow strategy is slower than the original one by 22.16%. The reinforced EVM with timestamp strategy is slower by 28.98%. The reinforced EVM with both strategies is slower by 32.95%. The time overhead mainly comes from the structure's maintenance and the traverse of each monitoring strategy when an interesting opcode encountered. As a conclusion, the time overhead of reinforced EVM* is concerned with the number of the monitoring strategies we added into the EVM and the contract size. Compared with the fuzzing time, the overhead is tolerable for financial critical applications.

D. Discussion

One potential threat to reinforced EVM* is the design of the strategy that used to monitor the vulnerabilities. The methodology we proposed just tells how to reinforce the original EVM, but the detailed strategy should be considered and designed seriously. Section III introduces the monitoring strategy used to test overflow and timestamp bugs. An inappropriate strategy may lead to some false alerts as well as some miss alerts.

Contract Number	Contract Lines	Original			Overflow			TimeStamp			Both		
		min (tx/s)	max (tx/s)	average (tx/s)	min (tx/s)	max (tx/s)	average (tx/s)	min (tx/s)	max (tx/s)	average (tx/s)	min (tx/s)	max (tx/s)	average (tx/s)
1	223 lines	3	11	3	2	4	3	2	3	2	2	5	2
2	8 lines	26	167	78	12	61	58	13	55	55	12	79	52
3	9 lines	13	69	69	15	84	50	12	53	50	22	96	46
4	44 lines	8	44	10	5	15	6	5	12	6	6	22	6
5	241 lines	3	16	3	3	5	2	2	4	2	1	5	2
6	1744 lines	2	12	2	2	4	2	1	3	2	2	5	2
7	442 lines	2	10	2	1	4	2	1	3	2	2	4	2
8	249 lines	5	26	5	6	22	8	2	7	4	7	27	9
9	286 lines	2	8	2	2	6	3	1	2	1	3	8	3
10	286 lines	2	8	2	2	6	3	1	2	1	3	8	3

Fig. 2. Time overhead of the transactions execution on the reinforced EVM*

Besides, the monitoring strategy has a great affect on the time overhead of the reinforced EVM. We also implement the reinforcement for three more types of bugs, the overhead will not increase too much with five strategies, because the main overhead is caused by the instrumentation and optcode collection, not by the constraint check.

V. Lessons Learned

During the design and implementation of the reinforced EVM*, we found out many lessons worthy to be discussed.

(1) EVM reinforcement could make up for the testing tools aimed at smart contract level. During the evaluation and industry practice, engineers found that the testing tools do have the disadvantages in missing certain vulnerabilities in the given program. They need different methods to ensure the security in more dimensions. In contrast, the reinforcement methodology does not focus on testing, but protecting the EVM from any potential dangerous operations in real time.

(2) EVM reinforcement could interrupt dangerous operations, not only those of smart contracts. Besides smart contracts, a portion of problems are due to EVM itself. Testing tools could only detect bugs in smart contracts to ensure the robustness of the program. However, if something went wrong in EVM which is irrelevant to smart contracts, reinforced EVM* could detect these problems while running and immediately interrupt the program.

(3) EVM reinforcement should be scalable and straightforward to be implemented on different EVMs. There are losts of different EVM platforms in industry, and the proposed approach should support different platforms. Luckily, based on the rules customized by Ethereum, each version of EVM platform is functionally and structurally the same. So that, the methodology we proposed could be easily implemented on other EVM platforms, following the common workflow.

VI. Conclusion

We proposed the framework EVM* to reinforce the EVM platform and protect the transactions from being attacked. Compared with the original EVM, the reinforced EVM* could stop the execution of dangerous transactions in real time. Besides, our work could make up for the testing tools. The testing tools may miss certain bugs while the reinforced EVM* could interrupt the transaction whenever finding a suspicious operation. For the time overhead, reinforced EVM* is slower than the original one by 20-30% in average. Our future work will focus on the exploration of the detecting strategies, and try to cover more types of bugs such as dangerous delegate call and reentrancy bugs. Furthermore, we will try to implement our methodology on more EVM versions.

References

[1] SmartContractSecurity, "Smart Contract Weakness Classification and Test Cases", https://github.com/SmartContractSecurity/SWC-registry, accessed 14-November-2018

[2] mythril-classic, "Mythril-classic", https://github.com/ConsenSys/mythril-classic/tree/develop/mythril/ethereum/interface/rpc, accessed 14-November-2018

[3] trailofbits, "Manticore", https://github.com/trailofbits/manticore, accessed 14-November-2018

[4] ethereumjs, "Number can only safely store up to 53 bits.", https://github.com/ethereumjs/ethereumjs-vm/issues/114, accessed 14-November-2018

[5] trailofbits, "echidna: Ethereum fuzz testing framework.", https://github.com/trailofbits/echidna, accessed 14-November-2018

[6] trailofbits, "The Ethereum Smart Contract Fuzzer for Security Vulnerability Detection", https://github.com/trailofbits/echidna, accessed 14-November-2018

[7] melonproject, "Oyente: An Analysis Tool for Smart Contracts.", https://github.com/melonproject/oyente, accessed 14-November-2018

[8] Luu L, Chu D H, Olickel H, et al. Making Smart Contracts Smarter[C]// ACM Sigsac Conference on Computer and Communications Security. ACM, 2016:254-269.

[9] Jiang B , Liu Y , Chan W K . ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection[J]. 2018.

[10] Serebryany K , Bruening D , Potapenko A , et al. AddressSanitizer: A Fast Address Sanity Checker[C]// Usenix Conference on Technical Conference. USENIX Association, 2012.

[11] Hildenbrandt E., Saxena M., Zhu X, et al. KEVM: A Complete Semantics of the Ethereum Virtual Machine// IDEALS, 2017.

[12] Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers[C]// International Conference on Financial Cryptography & Data Security. 2017.

[13] Yuanliang Chen, Yu Jiang, Jie Liang, and Mingzhe Wang. Enfuzz: From ensemble learning to ensemble fuzzing. arXiv preprint arXiv:1807.00182, 2018.

[14] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, and Chijin Zhou. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 809-814. ACM, 2018.

[15] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 562-566. IEEE, 2018.

[16] Mingzhe Wang, Jie Liang, Yuanliang Chen, and Yu Jiang. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pages 61-64. ACM, 2018.