Information Processing and Management 58 (2021) 102565

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/ipm

Security reinforcement for Ethereum virtual machine

Fuchen Ma [a,*], Meng Ren [a], Ying Fu [a], Mingzhe Wang [a], Huizhong Li [b], Houbing Song [c], Yu Jiang [a]

[a] Tsinghua University, China
[b] WeBank, China
[c] Embry-Riddle Aeronautical University, USA

ARTICLE INFO

Keywords:
Ethereum virtual machine
Online reinforcement
Vulnerability
Smart contract

ABSTRACT

Smart contracts are more sensitive from a security perspective than other software due to several reasons. First, smart contracts are immutable thus cannot be easily patched once deployed. Second, smart contracts are directly tied to payments and can hold millions of dollars' worth of digital currencies. Third, smart contracts are still a new practice thus do not have best coding practices and development lifecycles tailored for decentralized apps yet. Even though several testing and verification tools have been developed, smart contract vulnerabilities remain a clear and present danger. In this paper, we present an approach that is different from existing ones that attempt to eliminate vulnerabilities from smart contracts. Instead, we fortify Ethereum virtual machines (EVM) to stop dangerous transactions once vulnerabilities are detected in real-time. Since proving programs written in Turing-complete languages is undecidable, our approach complements current approaches by catching vulnerabilities and interrupts their executions during runtime. We have implemented our reinforcement on two widely used EVMs (js-evm and FISCO-BCOS-evm). The reinforced EVMs detects and interrupts all the vulnerabilities, 20% of them missed by testing tools, in 100 real smart contracts. Our approach is practical with less than 34% overhead. In fact, the reinforced FISCO-BCOS-evm has been integrated into the official release of FISCO-BCOS adopted by a large Chinese bank — WeBank.

1. Introduction

Ethereum is considered as the second large blockchain system in the world. Ethereum expands the blockchain concept with smart contracts. Smart contracts are the programs running on the Ethereum blockchain. In order to execute the smart contracts, Ethereum provides Ethereum Virtual Machine (EVM) to parse the source code of the contracts into an opcode sequence defined by Ethereum. Each node in Ethereum blockchain needs an EVM to execute the contracts properly and process the transactions. However, attacks on the smart contracts deployed on Ethereum platform are widespread and could cause a significant loss of money.

Many researchers attempted to improve the robustness of smart contracts with the customization of traditional testing techniques such as fuzzing and symbolic execution (melonproject, 2018; mythril-classic, 2018; trailofbits, 2018a). However, in real industry practice, we find that existing tools may miss some vulnerabilities hidden in the deep path of smart contracts. For example, we run ContractFuzzer on a contract with a known timestamp error for 2 h, but the bug was not detected. It is challenging for those techniques working on smart contract level to ensure security. The first is that testing before the deployment of smart contracts is not complete. It is not easy to explore all situations and paths, and the false positive and false negative cases would result in big

* Corresponding author.
E-mail address: mafc19@mails.tsinghua.edu.cn (F. Ma).

https://doi.org/10.1016/j.ipm.2021.102565
Received 22 June 2020; Received in revised form 31 January 2021; Accepted 27 February 2021

potential issues. Another challenge is that the contract cannot be altered once after being deployed on Ethereum, and testing tools of smart contracts cannot protect the deployed contracts.

In this paper, instead of working on the vulnerability detection at the smart contract level, we propose EVM* to reinforce the underlying EVM implementations, which could hunt and interrupt the dangerous transactions in real time. The reinforced EVM* consists of three steps: monitoring strategy definition, opcode-structure maintenance and EVM instrumentation. Monitoring strategy definition provides the detail constraints and rules to decide whether there is a dangerous operation such as integer overflow during the execution of transactions. Opcode-structure maintenance is to maintain a structure to store the interesting opcodes and parameters related to the strategy definition. EVM instrumentation is to insert the monitoring strategy, interrupting mechanism and the opcode-structure operations in the original EVM source code. Then, the reinforced EVM* could monitor all the transactions and stop dangerous transactions with the predefined interrupt mechanism in real time.

We need to solve two main challenges during the EVM* design and implementation. The first is to define the strategies accurately because incorrect monitoring strategies would result in serious false positives and false negatives. The second is to ensure the monitoring with tolerable overhead because huge overhead or resource consumption would limit the practical usage of the EVM*. Furthermore, EVM* should be scalable to different EVM implementations and vulnerability types.

For evaluation, we implement EVM* on two widely used EVMs: js-evm (ethereumjs, 2018a) which is implemented in JavaScript, and FISCO-BCOS-evm (http://www.fisco-bcos.org. (Accessed 23 August 2019)) which is implemented in C++. We implement four common monitoring strategies (integer overflow, timestamp dependency, delegatedcall to an untrusted callee and send with insufficient gas) and throw an exception when encountered an unsafe action. A stack is implemented to store the strategy related opcodes as well as the operands. 100 real world smart contracts are collected with known bugs. Then we made a dangerous transaction on each contract on the original EVM and the reinforced EVM*. **None of the dangerous transactions could be stopped by the original EVMs, while all the dangerous transactions on the reinforced EVM* could be interrupted successfully. For the time overhead, the reinforced EVM* with all the four monitoring strategies is slower than the original EVMs by 33.52%, and the reinforced EVM* with only one monitoring strategy ranges is slower for about 22.16%–28.98%.**

Our main contributions lay on the following aspects:

(1) We proposed a framework of reinforcing EVMs to prevent dangerous transactions in real time, which is scalable for different EVM platforms such as cpp-evm and js-evm, and different types of bug such as integer overflow and timestamp dependency error.
(2) We implemented the reinforced EVM* on two widely used EVMs, js-evm and FISCO-BCOS-evm to protect the transactions from 4 common types of smart contract vulnerabilities. The reinforced FISCO-BCOS-evm has been integrated into the official release version 2.0 of FISCO-BCOS[1] in WeBank Company.
(3) We evaluated the effectiveness of the original EVMs and the reinforced EVM*. The reinforced EVM* could successfully stop all dangerous transactions from execution with a tolerable time overhead.

2. Related work

In recent years, blockchain systems have been applied in various software systems, especially in information systems (Baniata et al., 2021; Berdik et al., 2021; Campanile et al., 2021; Chen et al., 2020b; Esposito et al., 2021; Hardin et al., 2020; Jing et al., 2021; Khalid et al., 2021; Li et al., 2020; Oham et al., 2021; Putz et al., 2021; Yu et al., 2021; Zhao et al., 2020). The security of the blockchain system has attracted the attention of many researchers. We discuss the most related work aiming at the vulnerability detection of the smart contracts and the EVMs.

Smart Contract Validation: Fuzzing and symbolic execution are useful for detecting vulnerabilities in traditional software. In order to find vulnerabilities in Ethereum smart contracts, researchers have developed many tools based on these two techniques. Fuzzing based tools contain: Echidna (trailofbits, 2018b) and ContractFuzzer (Durieux et al., 2019). Echidna finds out whether the invariants defined by the contract developers hold during the execution period of the contract. ContractFuzzer uses test oracles to detect security vulnerabilities. Symbolic execution tools contain: Oyente, Manticore and Mythril (Durieux et al., 2019). They can generate inputs which can trigger a critical path that may lead to a crash or property loss automatically.

Besides, there are also some work focusing on the vulnerabilities detection by static analysis. For example, some work (Grech et al., 2018; Ma et al., 2019) provide static analysis based tools that aim at detecting out-of-gas vulnerabilities in smart contracts. It uses a smart contract datalog analysis framework Vandal (Brent et al., 2018) to accomplish this goal. Securify (Tsankov et al., 2018) also uses datalog analysis to detect security problems in smart contracts. Other work (Amani et al., 2018; Bhargavan, Delignat-Lavaud, & Fournet, 2016; Grishchenko et al., 2018) models the contract through formal verification, and then verifies the logical integrity of smart contracts to find dangerous operations. A recent work (Hu et al., 2021) also use deep learning to detect malicious contracts.

EVM Validation: Some recent work (Fu et al., 2019) has proposed a new idea of vulnerability detection of Ethereum virtual machine via differential fuzzing. KEVM (Hildenbrandt et al., 2017) 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 (Hirai, 2017) provides a formal specification of the interface between smart contract execution and the rest of the world. Lem is a language designed

[1] The code of the current version of FISCO BCOS 2.0. is listed at https://github.com/FISCO-BCOS/FISCO-BCOS.

Table 1
Comparison between the vulnerability detection tools and EVM*.

Tools	Level	Method	Based on	Stage	Results
Echidna, ContractFuzzer trailofbits (2018b)	Contract	Fuzz	Transaction	Before deployed	Bug reports & Input seeds
Oyente, Manticore, Mythril Durieux et al. (2019)	Contract	Symbolic Execution	Control flow graph	Before deployed	Bug reports
MadMax, Securify Brent et al. (2018)	Contract	Static Analysis	Intermediate representation	Before deployed	Bug reports
TBCD Hu et al. (2021)	Contract	Deep Learning	Transaction	Before deployed	Bug reports
KEVM Hildenbrandt, Saxena, Zhu, et al. (2017)	EVM	Formal Verification	Intermediate representation	–	–
TAF Xu et al. (2021)	EVM	Model Validation	Transaction flow	–	Average transaction latency
EVM*	EVM	Transaction Monitoring	Transaction	Runtime	Transaction interruption

to compile to various interactive theorem provers, including Coq, Isabelle/HOL, and HOL4. This EVM definition can be used to prove invariants and safety properties of Ethereum smart contracts. Apart from security, latency performance is also crucial to a blockchain. Recently, (Xu et al., 2021) provides a novel theoretical model to calculate the transaction latency of Hyperledger Fabric under various network configurations.

Main Difference: EVM* is implemented based on our prior work (Ma, Fu, et al., 2019), with the support of more types of monitoring strategies and more types of EVM platforms. The main differences between our approach and the existing vulnerability detection tools are shown in Table 1.

As the table shows, the existing vulnerability detection tools detect the vulnerabilities on contract level, while EVM* secure the transactions on EVM level which is the same as KEVM and TAF. Different from the other tools, the method of EVM* is transaction monitoring. The analysis of EVM* is based on transactions while other tools are based on a CFG or certain IR. Other tools look for vulnerabilities before a contract is deployed, while EVM* monitors the dangerous transactions at runtime. The results of the detection tools are bug reports while EVM* interrupts all the detected dangerous transactions.

3. Motivating example

In this section, we illustrate the motivation of this work with a real contract. The code listed in Listing 1 shows an example of a lottery game.

The game chooses winners who have the *genius number*. If nobody has the *genius number*, there is no winner in this game. Function *choose_genius_number* randomly chooses a string as the *genius number* which is the hash value of the current timestamp. The second function *set_player_nums* set the number of a certain player. The number is chosen by the players as their own wishes. Function *reward* can only be called by the owner of the contract. This function checks each number of players then it chooses the winners and rewards them.

```
1  contract GeniusLottery{
2      uint genius_number;
3      address[] players;
4      mapping(address => uint) public player_num;
5      mapping(address => uint) public balance;
6      function choose_genius_number() public
7          {genius_number = sha256(now);}
8      function set_player_nums
9          (address player, uint num) public
10         {player_num[player] = num;}
11     function reward() public{
12         for(uint i = 0;i < players.length;i++){
13             if(player_num[players[i]] == genius_number)
14                 balance[players[i]] += 2000;
15                 balance[msg.sender] -= 2000;
16         }}
17 }
```

Listing 1: An example contract with an overflow bug, however, it is hard for dynamic testing tools to detect this bug.

As we could see at line 21 and line 22, it is unsafe to use an add operation or a subtraction operation without checks. The contract may be greatly influenced by overflow if the balance of the player is bigger than the maximum value that a *uint* variable can represent or the balance of the sender is less than *2000*. Fuzzing based tools aimed at smart contracts can hardly trigger the bug of line 21 and line 22, because this bug is closely related to the constraint of line 19–20 and line 15, which is hard to be solved by existing search or symbolic engine. Fig. 1 shows the control flow graph of the function *reward*. Because it is really hard to generate a seed where the number of some player happen to be the hash value of the genius number, and the branch *e* in the figure is hard to be covered.

However, in practical application, branch *e* may be commonly executed. For example, the number that the players could choose may be limited to the sponsor of the lottery. Unfortunately, fuzzing tools have no knowledge about this scope of number. They can

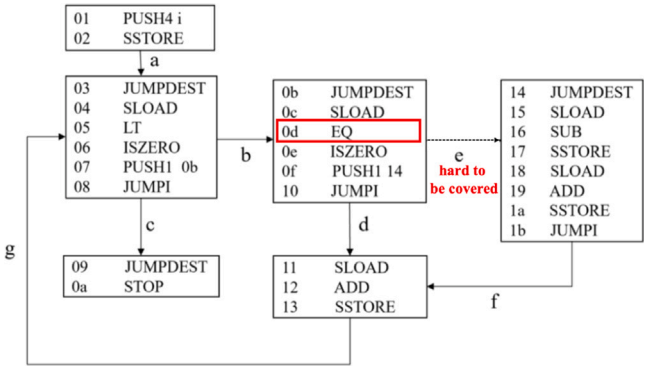

Fig. 1. The control flow of function *reward* on bytecode code level. According to the result of the opcode 'EQ', the branch e is hard to be covered by dynamic testing tools which leads to a false-negative of the tools.

only choose the number in a relatively random way which leads to the miss of this vulnerability with high probability. Furthermore, even if we are lucky enough to detect the bug during the testing procedure, how can we protect the transaction, because most deployed applications cannot be revised. It is not easy to solve the problems on the source code of smart contract, so we try to work on the EVM level.

The reinforced EVM* can monitor and interrupt the above overflow bug with the following three steps. First, EVM* append the operators such as "ADD" and "SUB" into the list of *interesting operators*. Then, before the execution of each *interesting operator*, EVM* will check the opcode stack with the monitoring strategies. The monitoring strategy is the rule defined to check whether the current opcode stack is dangerous or not. EVM* has a unique monitoring strategy for each type of vulnerability. For the integer overflow vulnerability, EVM* will check all the operators of the numeric operators, i.e. "MUL", "ADD" and "SUB", and judge whether the results of the operator under current operators have overflow situation, i.e. the addition result of two positive numbers turns out to be negative. In this way, EVM* is able to sensor the overflow during run-time execution and stop the dangerous transaction immediately, and protect those transactions from being affected by contract vulnerabilities.

4. Reinforcement methodology EVM*

Different from existing work trying to detect vulnerabilities at the smart contract level, EVM* applies run-time verification to monitor and interrupt dangerous transactions at the EVM level. The reinforced EVM* contains three main components, as presented in Fig. 2. The monitoring strategy component refers to the definition that whether an opcode sequence is dangerous and how to stop a dangerous transaction. Engineers can define the rules for the vulnerabilities they want to support. The opcode-structure maintenance component contains three actions: initialization, record and analysis for run-time decision. The EVM instrumentation component inserts the monitoring strategies and structure operations into the original EVM source code, resulting in the reinforcement EVM*. The input of EVM* is the smart contract byte-code and the transaction data.

4.1. Monitoring strategy

Monitoring strategy is the most important part of the reinforced EVM*, because the accuracy of the strategy is related to the false positive and negative of the dangerous transaction detection. The strategy consists of two parts: interesting opcodes and the constraints for the opcode sequences. We show how to define the monitoring strategies with four common types of vulnerabilities in smart contracts: overflow bugs, timestamp dependency bugs, delegatecall to untrusted callee bug and send with insufficient gas bug.

4.1.1. Integer overflow

Variables of numerical type in smart contracts have legal interval. If the variable exceeds the maximum value, it will circle back to zero. This potential threat is called integer overflow, which could lead to a big loss of money. Although there is a safe way to avoid overflow situations with the help of *SafeMath* library, which provides safe numeric computing operators, there are still many attacks and dangerous transactions based on this vulnerability.

As Listing 2 shows, the first contract has a function named 'add'. The variable balance in the first contract is defined by the type uint256. If the variable *addBalance* given by the user is big enough, the balance will possibly be bigger than 256 power of 2. The balance will be decreased though it was actually added with a big positive value. This is the most obvious impact of overflow errors, which could obviously lead to a money loss. In the second contract, the variable 'i' is defined by the key word 'var', which represents 8-bit binary numbers. Hence, the maximum value of 'i' is 255. If the length of the array 'account' is bigger than 255, the loop will never stop. This may cause a denial-of-service attack due to the out-of-gas error.

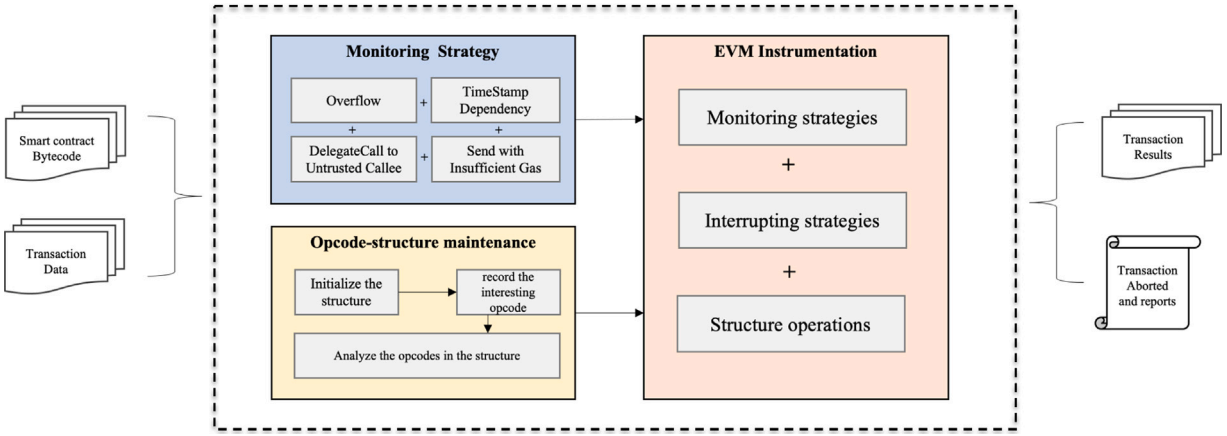

Fig. 2. The framework of the reinforced EVM*. It mainly consists of three components: Monitoring strategy for run-time bug detection, Opcode-structure maintenance for operation monitoring, and EVM instrumentation for transaction decision and protection. The output of the virtual machine is either the transaction results or the aborted report of the dangerous transaction.

```
contract Overflow_1{
    uint256 balance;
    function add(uint256 addBalance)
       { balance = balance + addBalance;}
}
contract Overflow_2{
    Account account[];
    function goOver(){
        for(var i = 0; i < account.length(); i++)
        {...}
    }
}
```

Listing 2: Two integer overflows in smart contracts. The first may lead to property losses and the second may lead to an out-of-gas exception.

Detection strategy. To infer whether an overflow action occurs in a transaction, the interesting opcodes contains 'ADD', 'SUB', 'MUL', 'ADDMOD', 'MULMOD' and 'EXP'. The constraints that could be used to detect overflow vulnerability contains the following situations:

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

$$\{value(op1) > 0 \ \cap value(op2) > 0$$
$$curOp(\text{“}ADD\text{”}) \cap value(res) < 0\}$$

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

$$\{value(op1) < 0 \ \cap value(op2) < 0$$
$$curOp(\text{“}ADD\text{”}) \cap value(res) > 0\}$$

- If a positive number subtracts a negative number, but the result is negative.

$$\{value(op1) > 0 \ \cap value(op2) < 0$$
$$curOp(\text{“}SUB\text{”}) \cap value(res) < 0\}$$

- If a negative number subtracts a positive number, but the result is positive.

$$\{value(op1) < 0 \ \cap value(op2) > 0$$
$$curOp(\text{“}SUB\text{”}) \cap value(res) > 0\}$$

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

$$\{value(op1) > 0 \ \cap value(op2) > 0$$
$$curOp(\text{“}MUL\text{”}) \cap value(res) < 0\}$$

- If two negative numbers perform an multiplication operation, but the result of the operation is negative.

$$\{value(op1) < 0 \ \cap value(op2) < 0$$
$$curOp(\text{“}MUL\text{”}) \cap value(res) > 0\}$$

value() represents the value of the operator, and *curOp*() represents the type of the operator. If any of the above formulated constraints is violated, an integer overflow alarm would be reported.

4.1.2. Timestamp dependency

Every Ethereum block contains a timestamp which smart contracts can read, and timestamps are often used as the basis for generating random numbers. Unfortunately, the timestamp can be manipulated by the miner who is not always a user's friend. The miner can adjust the timestamp provided by a few seconds, thus changing the output of the contract to his own benefit. For example, in some lottery contracts, the winning ticket number is randomly generated. However, in Ethereum, miners can modify the timestamp within a certain range. To a certain extent, this allows miners to control the generation of random numbers, and then control the winning probability. Similar vulnerabilities include block number dependency.

```
1  pragma solidity ^0.5.0;
2  contract UseTimeStamp {
3      address currentAddress;
4      uint someVariable = now + 1;
5      if (now % 2 == 0) {
6      // This variable can be manipulated by miners.
7          currentAddress.send(100);
8      }
9      if ((someVariable - 100) % 2 == 0) {
10     // someVariable can be manipulated by miners.
11         currentAddress.send(200);
12     }
13 }
```

Listing 3: A contract that use timestamp to determine the amount of send operation. The miner can manipulate the timestamp to control the value.

The code presented in Listing 3 is a contract with timestamp dependency bugs. From line 4, 5 and 10, we can see that the contract uses the timestamp to determine the value of the send operation, and the miners can manipulate the timestamp to control the sending process. If maliciously exploited, money would lost.

Detection strategy. The interesting opcode that timestamp bugs concerned about is 'TIMESTAMP'. If a transaction contains 'TIMESTAMP' opcode we then checks the value of it. 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 is a timestamp bug which is possibly exploited by malicious miners. The strategy of this type of vulnerability is shown as the following:

$$\{t | \exists call : t \ value(call) > 0 \cap$$
$$exist(\varepsilon TIMESTAMP\varepsilon, call)\}$$

t represents the current transaction, and *call* refers to each call in the transaction.

4.1.3. Delegatecall to untrusted callee

Delegatecall is a special type of message call which is identical from the normal function call except the fact that the targeted code is executed with the context of the calling contract. Using delegatecall to execute an untrusted contract code is extremely dangerous because the malicious users could get the storage variables of the caller easily in that context. The code listed below is an example of the contract with dangerous delegatecall vulnerabilities.

```
1  contract Proxy {
2    address owner;
3    constructor() public {owner = msg.sender;}
4    function forward(address callee,bytes _data) public {
5      require(callee.delegatecall(_data));}
6  }
```

Listing 4: A contract uses a delegatecall to an untrusted callee. The calling can expose the whole contract to the untrusted account.

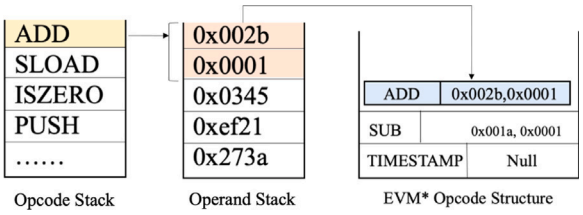

Fig. 3. The stack opcode structure of EVM*. Each element of the code is the combination of the opcode and its operands.

The code presented in Listing 4 includes a function 'forward', which allows an arbitrary callee to call some message under the context of the current contract. If malicious exploited, there is no privacy, and all data would be leaked for further attack.

Detection strategy. The interesting opcode that dangerous delegatecall error concerns is 'DELEGATECALL'. We check whether there is a delegatecall during the execution of the contract and whether the parameters of the delegatecall are related to the root call. If this happens, delegatecall may call malicious code and modify some key attributes of the contract itself. The strategy of this type of vulnerability is shown as the following:

$$\{t | \exists call : t\ rootInput.index(curInput) >= 0 \cap exist(\varepsilon DELEGATECALL\varepsilon, call)\}$$

rootInput represents the parameters of the root call and *curInput* refers to the parameter of the current call.

4.1.4. Send with insufficient gas

A send operation in Ethereum always calls the fallback function of the called contract. The gas limit of the callback function is usually 2300 units of gas. However, if the callback function is complex and costs more gas, the send operation will fail due to the out-of-gas exception. If the exception is not checked correctly, a malicious sender may reserve the ether in such method.

```
1 contract Lottery {
2   mapping(uint => address) public winners;
3   function givePrize(address[] winners){
4     for(uint i = 0; i < winners.length(); i++){
5         winners[i].send(1000);
6     }
7   }
8 }
```

Listing 5: A contract with a send which has insufficient gas. The out-of-gas exception may block the prize sending to other winners.

The code presented in Listing 5 shows an example of lottery contract, which contains an insufficient send threat. The lottery has more than one winner, each of which will be rewarded with a prize. However, as shown at line 6, if the callback function of some winner is computation-heavy, the out-of-gas error will block the rewarding to other winners. A safe way to use send is to check whether an out-of-gas happens and handle it correctly.

Detection strategy. The interesting opcode that this error concerns is 'CALL'. An transaction with the opcode 'CALL' and a gas limit of 2300 units is regard as a send operation. An insufficient gas send error happens when the send operation throws an out-of-gas exception. The strategy for this type of vulnerability is shown as the following:

$$\{t | \exists call : t\ call.OutOfGas() \cap call.gas == 2300\}$$

OutOfGas means that there is an out-of-gas exception in the current call. Meanwhile, the gas used is exactly 2300 units when the exception occurs.

4.2. 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. The candidate structures for the implementation of opcode-structure could be stack, queue and tree. Generally speaking, a stack structure has the following advantages. First of all, stack is easy to be implemented and maintained. Also, EVM is a stack based system, and using a stack structure as an opcode structure can facilitate the acquisition of some state information of the EVM.

Fig. 3 shows the stack structure for the monitoring of interesting opcodes. The element in the EVM* stack is the combination of the opcode and its operands. Suppose the current opcode is 'ADD'. The current state of the stack is shown in the figure. 'ADD' pops two elements from the stack and push the result back to the stack. So, the current operands are *0x002b* and *0x0001*. EVM* combines the opcode and its operands and push the combination into the opcode stack for the runtime analysis.

4.3. EVM instrumentation

EVM instrumentation module implements the monitoring strategy and interrupting mechanism in the original EVM source code.

Algorithm 1: Instrumentation process

```
   Input: stg: Monitoring strategies.
1  Function instrument(stg):
2      stacks = [];
3      op_Stack = new OP_STACK();
4      for op in opcodes do
5          if op.isCALL() then
6              if !op_Stack.isEmpty() then
7                  stacks.append(op_Stack);
8                  op_Stack.clear();
9                  op_Stack.push(new Item(CALL, op.operands()));
10             end
11         end
12         else if op.isInteresting() then
13             name = op.name();
14             op_Stack.push(new Item(name, op.operands()));
15             for s in stg do
16                 if !Test(s, stacks, op_Stack)) then
17                     Interrupt();
18                 end
19             end
20         end
21         execute(op);
22     end
23 End Function
```

The overall instrumentation process is shown in Algorithm 1. The input of the instrumentation process is the monitoring strategies. As line 3 shows, a new stack named op_Stack will be initialized to store the opcodes before the instrumentation. For each opcode of the opcode sequence, if it is a CALL opcode, the reinforced EVM* will check whether the op_Stack is empty. If not, the op_Stack will be stored in the list 'stacks'. After that EVM* will push the opcode 'CALL' and its operands into stack. This process is described from line 5 to line 11. In addition, if the current opcode is an interesting one, as illustrated by line 12–14, the reinforced EVM* will push it into the stack. Line 15–19 demonstrates that the current stack will be tested under each monitoring strategy. If the test result shows it is a dangerous operation, the execution is interrupted. Otherwise, the opcode will be executed successfully.

4.3.1. Instrumentation on js-evm

We describe the detail instrumentation of four monitoring strategies for the common types of vulnerabilities on js-evm.

Integer Overflow. The code listed in Listing 6 is used for the overflow monitoring of the add operation in js-evm. The function 'ADD' is used to add two numbers and use a suitable variable to store the result. However, the *uint* type in the solidity language defaults to a maximum of 256 bits. If the result is bigger than the maximum value that a 256 bits integer could represent, there would be an overflow. If there is an overflow, the execution will stop immediately and the reinforced EVM* will throw an exception.

```
ADD: function (a, b, runState) {
    var res = a.add(b)
    if (detectOverflow(res.toString())) {runState.stopped = true; throw "overflowbug"}
}
```

Listing 6: Code used to detect overflow on add operation.

The detail of function 'detectOverflow' is defined as Listing 7. The variable 'max' is defined as the 256 power of 2. If the result is bigger than 'max', an overflow occurs. Otherwise, there is no overflow. Other situations defined in Section 4.1.1 can be implemented and instrumented in the similar way.

```
function detectOverflow (res) {
  // We use 2^256 here for space limitation.
  var max = 2^256
  if (res.length > max.length) {return true}
  if (res.length < max.length) {return false}
  if (res >= max) {return true}
  return false
}
```

Listing 7: Code used to judge overflow in js-evm.

Timestamp Dependency. The code instrumented to monitor the timestamp dependency bug is presented in Listing 8 and Listing 9. The first piece of code is used to detect TIMESTAMP opcode. This function transfers all of the opcodes in the current call into a string, and checks whether the word 'TIMESTAMP' is in the string. The instrumented code presented in Listing 9 is used to detect whether there is any timestamp dependent Ether transfer during the transaction execution.

```
TimestampOp.prototype.TestTimestamp =
    function () {
    var rootCall = this.evmStar_calls[0]
    if ((rootCall.OperationStack.toString()).
        indexOf('TIMESTAMP') >= 0) {
      //analyze if "TIMESTAMP" occurs
      return true
    }
    return false
  }
```

Listing 8: Code used to detect whether there is a 'TIMESTAMP' opcode.

The first function named 'TestEtherTransfer' is used to check the sequence of call. The variable 'calls' records the current call as well as the related calls based on the current call. For each one in the sequence, the second function 'FocusCall' will test whether the call has some Ether transfer. If any call in the sequence has some ether transfer, the current call is not safe. When both constraints are satisfied, there is a timestamp dependent error in the transaction, and should be interrupted.

```
EtherTransfer.prototype.TestEtherTransfer =
    function () {
    var ret = false
    var calls = this.reinforce_calls[0].nextcalls
    for (var i = 0; i < calls.length; i++) {
      var call = calls[i]
      if (this.FocusCall(call)) {
        ret = true
      }
    }
    return ret
}
EtherTransfer.prototype.FocusCall =
    function (call) {
    return call.value > 0
}
```

Listing 9: Code used to detect whether there is Ether transfer.

Delegatecall to Untrusted Callee. The instrumented code used to monitor the vulnerability of delegatecall to untrusted callee is presented in Listing 10. The first function 'GetInput' checks whether some parts of the root call is given as the input of the delegatecall. The second function 'Delegatecall' detects whether there is a 'DELEGATECALL' opcode in the transaction.

```
ReinforceDelegateCallInfo.prototype.GetInput = function (rootcall, call) {
    if (rootcall.input.indexOf(call.input) >= 0) { return true }
    return false
}
ReinforceDelegateCallInfo.prototype.DelegateCall = function (call) {
    if (call.OperationStack.toString().indexOf('DELEGATECALL') >= 0) {
      return true}
    else {return false}
}
```

Listing 10: Code used to detect 'DELEGATECALL' opcode and whether the input of delegatecall is dangerous.

The code listed at Listing 11 shows the constraints. The function checks whether the current call is a delegate call or not. If it is, the function checks whether the input of the delegate call is dangerous. If both conditions are satisfied, there is a dangerous delegatecall bug in this transaction.

Send with Insufficient Gas. The instrumented code used to monitor the vulnerability of send with insufficient gas is presented in Listing 12. The function 'Test' checks each call in the call sequence and test each one with the function 'ExceptionSend'. If any

one has an insufficient gas send, the transaction will be stopped. The function 'ExceptionSend' in line 15 checks whether there is an out-of-gas exception in a call and whether the call is a send operation. There are two conditions need to be satisfied if the call is a dangerous send operation. As line 17 shows, there is no input for a send operation and the gas used when the out-of-gas error occurs is 2300 units.

```
1  ReinforceDelegateCallInfo.prototype.Test
2     = function () {
3     var hasDelegate = false
4     var nextcalls = this.reinforce_calls[0].nextcalls
5     for (var i = 0; i < nextcalls.length; i++) {
6       var call = nextcalls[i]
7       if (this.TriggerDelegateCall(call) &&
8         this.GetFeatures(this.reinforce_calls[0]
9             , call)) {
10        this.reinforce_delegate_calls.push(call)
11        hasDelegate = true
12      }
13    }
14    return hasDelegate
15  }
```

Listing 11: Code used to detect dangerous delegatecall.

```
1  ReinforceGaslessSend.prototype.Test
2     = function () {
3     var hasException = false
4     var calls = this.reinforce_calls[0].nextcalls
5     for (var i = 0; i < calls.length; i++) {
6       var call = calls[i]
7       if (this.ExceptionSend(call) == true) {
8         this.reinforce_exception_calls.push(call)
9         hasException = true
10
11      }
12    }
13    return hasException
14 }
15 ReinforceGaslessSend.prototype.ExceptionSend
16    = function (call) {
17    return call.gasException == true &&
18    call.input == '' && call.gas == 2300
19 }
```

Listing 12: Code used to monitor insufficient gas send vulnerability. For each call, the function 'Test' checks whether it is a send operation and throws an out-of-gas exception.

The interrupt mechanism is instrumented by using the exception handling of javascript. To be specific, during the execution of the opcodes, the reinforced EVM checks whether there is an exception in the execution. The monitoring strategy will throw an exception which contains the vulnerability type and call stack information. The following opcodes of the dangerous transaction will then be blocked from executing. In consequence, the transaction will be reverted and the attacks will not succeed in this way.

4.3.2. Instrumentation on FISCO-BCOS-evm

The instrumentation on the virtual machine of FISCO-BCOS-evm is the same as the one on js-evm except for the integer overflow. The difference for integer overflow vulnerability monitoring is due to the different representation of number in two virtual machines. In js-evm implemented in JavaScript, the number is represented as a string type variable, while in FISCO-BCOS-evm implemented in C++, the number is represented as an integer. The code to monitor integer overflow vulnerability for FISCO-BCOS-evm is shown as Listing 13. The code first checks whether the two numbers are with the same symbol. If the two numbers with different symbols are added together, there will be no overflow. If there are two positive numbers, we use an integer with 512 bits to receive the result of the add operation. Meanwhile, we will also use an integer with 256 bits to receive the result. If the results are the same, there is no overflow here. Otherwise, an overflow occurs. Others situations defined in Section 4.1.1 can be implemented and instrumented in the similar way.

```
CASE(ADD){
    ON_OP();
    updateIOGas();
    // pops two items and pushes their sum.
    u256 a = m_SP[0];
    u256 b = m_SP[1];
    if (((u2s(a) >= 0) && (u2s(b) <= 0)) ||
        ((u2s(a) <= 0) && (u2s(b) >= 0))) {
        m_SPP[0] = m_SP[0] + m_SP[1];
    } else{
        if (u2s(a) >= 0 && u2s(b) >= 0){
          u256 temp = a + b;
          u512 temp_ = u512(a) + u512(b);
          if (temp != temp_){
             throw "SOL_ASANCrash";}
          else{m_SPP[0] = m_SP[0] + m_SP[1];}
        }else{
            a = s2u(0 - u2s(a));
            b = s2u(0 - u2s(b));
            u256 temp = a + b;
            u512 temp_ = u512(a) + u512(b);
            if (temp != temp_){
             throw "SOL_ASANCrash";}
            else{m_SPP[0] = m_SP[0] + m_SP[1];}
        }
      }
}
```

Listing 13: Code used to monitor overflows on 'ADD' opcode in the virtual machine of FISCO-BCOS-evm.

5. Evaluation

In our evaluation of the reinforced EVM*, we will answer the following research 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?

5.1. Data and environment setup

We implemented the proposed security reinforcement EVM* on two virtual machines: js-evm and FISCO-BCOS-evm. Js-evm is widely used and developed by JavaScript. FISCO BCOS (http://www.fisco-bcos.org. (Accessed 23 August 2019)) is an alliance chain platform specially designed for the financial applications by WeBank company, and has attracted many attentions in blockchain industry. FISCO-BCOS-evm is based on the EVM implemented in C++. The version of the solc compiler we used is 0.4.25, and the operating system is Linux x86_64.

To answer the first question, we embedded the 4 types of vulnerabilities into 100 contracts[2] for evaluation. We used each contract to execute a dangerous transaction both on the original EVM and the reinforced EVM*. Furthermore, we developed and initialized a simple fuzzing environment based on ContractFuzzer to imitate the transaction process. In a transaction, each function will be executed random times with arbitrary parameters. The environment augmented with the ability to detect four types of bugs, will create lots of transactions, to see whether these bugs could be detected, in the duration time of 1 h in this experiment.

To answer the second question, we made transactions on the original EVM and reinforced EVM* with random inputs. We calculated the amount of transactions made in the time limit on both versions to evaluate the time overhead caused by the reinforcement.

5.2. Effectiveness of the reinforced EVM*

We first used the ContractFuzzer (Jiang, Liu, & Chan, 2018) and sFuzz (Nguyen et al., 2020) to test all of the contracts with embedded bugs. The results are shown as Table 2. There are 25 vulnerable contracts for each type. As the data shows, 80% of the vulnerable contracts can be detected by ContractFuzzer while only 28% can be detected by sFuzz, the missed bugs can still result in potential dangerous situations.

[2] The dataset can be accessed at: https://github.com/EVM-Reinforcement/EvaluatingContracts.git.

Table 2
The results of ContractFuzzer and sFuzz on the 100 contracts with embedded bugs. Only 80% of the contracts can be detected by contractFuzzer and 28% of the contracts can be detected by sFuzz.

Tools	Overflow	TimeStamp dependency	Dangerous DelegateCall	Insufficient Gas Send
ContractFuzzer	17	19	21	23
sFuzz	5	15	8	0

Table 3
The effectiveness of reinforced EVM* w.r.t. original EVM, to stop dangerous transactions.

Bug Type	Number of contract	Original EVM	Reinforced js-evm	Reinforced FISCO-BCOS-evm
Overflow	25	0	25	25
TimeStamp dependency	25	0	25	25
Dangerous DelegateCall	25	0	25	25
Insufficient Gas Send	25	0	25	25

Among those contracts missed by the fuzzing tool, we analyzed one of them named overflow_simple_add.sol. The source code of the contract is presented in Listing 14.

```
contract Overflow_Add {
    uint public balance = 1;
    function add(uint256 deposit, uint256 value){
        if(prefix(sha256(value)) == "01001001001")
          balance += deposit;
    }
}
```

Listing 14: A contract with an overflow vulnerability that cannot be detected by fuzz testing or symbolic execution tool. However, the reinforced EVM* can stop the dangerous transaction if an overflow occurs.

This contract has only one function named ‘add’, and this function receives an input named ‘deposit’ and another input named ‘value’. The type of both parameters is uint256. In the function ‘add’, if the hash value of the parameter has the exactly prefix as the fixed one (“01001001001” in this case), the current balance will increase. The variable ‘balance’ is added with the input variable deposit and the result will be stored in the variable ‘balance’.

However, because the value of variable ‘balance’ is 1, only if deposit’s value is the maximum number that a uint256 could represent, the overflow would occur. Besides, it is really hard for existing fuzzing or symbolic execution tools to generate a seed that could satisfy the hash prefix condition which makes it even harder for fuzzing tool to detect this vulnerability.

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 existing fuzzing tools and symbolic execution tools that they could not detect some bugs that will occur in some extreme cases. However, reinforced EVM* is able to monitor the bug in real time and stop the transaction timely. Table 3 presents the detail experimental results of the reinforced EVM* compared with the original EVM to stop the dangerous transactions.

Last three columns in the table represent the number of contracts that could be stopped when the dangerous operation occurs. From the table, we found that for the original EVM, all the dangerous transactions have been permitted. However, all the dangerous transactions have been stopped from executing by the reinforced EVM*.

From the result, it is reasonable to draw the conclusion that the reinforced EVM* could successfully monitor and stop the dangerous transactions from executing, which could effectively ensure the security of the transaction, even when the deployed contracts have vulnerabilities.

*5.3. Time overhead of the reinforced EVM**

We count the total transactions made during the experiment of each contract. The transaction here refers to the collection of calls to each function in the contract. The results are shown in Table 4. We use reinforced EVM* with four types of monitoring strategies in the evaluation. In order to calculate the execution time, we delete the interrupting mechanism in the EVM* and reserve the other code. In this way, EVM* will also check the transaction with each strategy but will not stop the transaction.

The unit of the number in the first row of the table is counted by transactions per second, which is abbreviated as tx/s. We calculate the transactions made with six types of EVM: the original EVM, the EVM* with overflow strategy, the EVM* with timestamp

Table 4
Transactions made in a second by the original EVM, EVM* with one of the four strategies and EVM* with all of the four strategies.

	Original EVM	EVM with overflow strategy	EVM with timestamp strategy	EVM with Dangerous DelegateCall strategy	EVM with Insufficient Gas Send strategy	EVM with four strategies
Transactions per sencond	17.6	13.7	12.5	13.4	12.8	11.7
Slower by	–	22.16%	28.98%	23.86%	27.27%	33.52%

strategy, the EVM* with dangerous delegatecall strategy, the EVM* with insufficient gas send strategy and the EVM* with four strategies. For each version, we calculate the average amount of transactions made per second on each contract.

As the result shows, 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 dangerous delegatecall strategy is slower by 23.86%. The reinforced EVM* with insufficient gas send strategy is slower by 27.27%. The reinforced EVM with four strategies is slower by 33.52%.

5.4. Discussion

One potential threat to the experiment results is the collection of experimental data. In the process of dealing with experimental data, there are often omissions and errors. In order to avoid this problem, we use fully automated scripts to integrate experimental data and count the true-positive samples, avoiding the effect of human error to this experiment. During the design and implementation of the reinforced EVM*, we also 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. Different from the smart contract analysis tools, 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 also capture these problems during running and immediately interrupt the program.

(3) EVM reinforcement framework can enhance the performance and security of blockchain information systems. Recently, many works integrated the concept of blockchain into the information systems. Q Chen, et al. (Chen et al., 2020a) proposes an approach using a novel blockchain-based solution for IoFMT incorporated with a gamification component. Q Zhao, et al. (Zhao et al., 2020) utilize the blockchain to construct a novel privacy-preserving remote data integrity checking scheme for Internet of Things (IoT) information management systems. Our approach can replace the virtual machined used in those blockchain system. The execution of the transactions will be secured with the reinforced EVM*.

6. Conclusion

In this work, we proposed the framework EVM* to protect the transactions on Ethereum from being attacked. The reinforced EVM* can make up for the current testing tools and stop the execution of dangerous transactions in real time. For the time overhead, reinforced EVM* is slower than the original one by 20%–30% in average. EVM* could still be used to monitor some other vulnerabilities which are not only from the execution of smart contracts. Furthermore, the proposed framework has been proved to use on different blockchain platform. EVM* has some practical values which could help the developers to build the platform better, and has been integrated in WeBank Company.

Our future work will focus on the exploration of the detecting strategies, and try to cover more types of bugs such as out-of-gas bugs. Besides, we will reduce the time overhead caused by the reinforcement by optimizing the monitoring strategy and implementation of the opcode structure. Besides, we will try to implement our methodology on more EVM versions.

CRediT authorship contribution statement

Fuchen Ma: Conceptualization, Methodology, Writing - original draft, Software. **Meng Ren:** Software, Data curation. **Ying Fu:** Software, Validation. **Mingzhe Wang:** Software, Investigation. **Huizhong Li:** Supervision. **Houbing Song:** Validation, Writing - reviewing. **Yu Jiang:** Supervision, Writing - reviewing.

Acknowledgment

This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200).

References

Amani, Sidney, et al. (2018). Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *CPP 2018*.

Baniata, Hamza, et al. (2021). PF-BTS: A Privacy-Aware Fog-enhanced Blockchain-assisted task scheduling. *Information Processing & Management*, *58*(1).

Berdik, David, et al. (2021). Survey on blockchain for information systems management and security. *Information Processing & Management*, *58*(1), Article 102397.

Bhargavan, K., Delignat-Lavaud, A., & Fournet, C. (2016). Short paper: Formal verification of smart contracts ACM 2016 article.

Brent, Lexi, et al. (2018). Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981.

Campanile, Lelio, et al. (2021). Designing a GDPR compliant blockchain-based IoV distributed information tracking system. *Information Processing & Management*, *58*(3), Article 102511.

Chen, Qian, et al. (2020a). An incentive-aware blockchain-based solution for internet of fake media things. *Information Processing & Management*, *57*(6), Article 102370.

Chen, Qian, et al. (2020b). An incentive-aware blockchain-based solution for internet of fake media things. *Information Processing & Management*, Article 102370.

Durieux, Thomas, et al. (2019). Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. arXiv preprint arXiv:1910.10601.

Esposito, Christian, et al. (2021). Blockchain-based authentication and authorization for smart city applications. *Information Processing & Management*, *58*(2), Article 102468.

ethereumjs (2018a). Number can only safely store up to 53 bits. https://github.com/ethereumjs/ethereumjs-vm/issues/114. (Accessed 14 November 2018).

Fu, Ying, et al. (2019). Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*.

Grech, Neville, et al. (2018). Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, *2.OOPSLA*, 116.

Grishchenko, Ilya, et al. (2018). A semantic framework for the security analysis of ethereum smart contracts. ArXiv abs/1802.08660, n. pag.

Hardin, Taylor, et al. (2020). Amanuensis: Information provenance for health-data systems. *Information Processing & Management*, *58*(2), Article 102460.

Hildenbrandt, E., Saxena, M., Zhu, X., et al. (2017). *KEVM: A complete semantics of the ethereum virtual machine*. IDEALS.

Hirai, Y. (2017). Defining the ethereum virtual machine for interactive theorem provers. In *International conference on financial cryptography & data security*.

Hu, T., et al. (2021). Transaction-based classification and detection approach for Ethereum smart contract. *Information Processing & Management*, *58*(2), Article 102462.

Jiang, Bo, Liu, Ye, & Chan, W. K. (2018). Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM international conference on automated software engineering*. IEEE.

Jing, Nan, et al. (2021). A blockchain-based code copyright management system. *Information Processing & Management*, *58*(3), Article 102518.

Khalid, Adia, et al. (2021). A blockchain based incentive provisioning scheme for traffic event validation and information storage in VANETs. *Information Processing & Management*, *58*(2), Article 102464.

Li, Jiaxing, et al. (2020). Blockchain-based public auditing for big data in cloud storage. *Information Processing & Management*.

Ma, Fuchen, Fu, Ying, Ren, Meng, Wang, Mingzhe, Jiang, Yu, Zhang, Kaixiang, et al. (2019). EVM*: From offline detection to online reinforcement for ethereum virtual machine. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering*. IEEE.

Ma, Fuchen, et al. (2019). Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. arXiv preprint arXiv:1910.02945.

melonproject (2018). Oyente: An analysis tool for smart contracts. https://github.com/melonproject/oyente. (Accessed 14 November 2018).

mythril-classic (2018). Mythril-classic. https://github.com/ConsenSys/mythril-classic/tree/develop/mythril/ethereum/interface/rpc. (Accessed 14 November 2018).

Nguyen, Tai D., et al. (2020). sFuzz: An efficient adaptive fuzzer for solidity smart contracts. arXiv preprint arXiv:2004.08563.

Oham, Chuka, et al. (2021). B-FERL: Blockchain based framework for securing smart vehicles. *Information Processing & Management*, *58*(1), Article 102426.

Putz, Benedikt, et al. (2021). EtherTwin: Blockchain-based secure digital twin information management. *Information Processing & Management*, *58*(1), Article 102425.

trailofbits (2018a). Manticore. https://github.com/trailofbits/manticore. (Accessed 14 November 2018).

trailofbits (2018b). echidna: Ethereum fuzz testing framework. https://github.com/trailofbits/echidna. (Accessed 14 November 2018).

Tsankov, Petar, et al. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. ACM.

Xu, Xiaoqiong, et al. (2021). Latency performance modeling and analysis for hyperledger fabric blockchain network. *Information Processing & Management*, *58*(1), Article 102436.

Yu, Guangsheng, et al. (2021). A novel Dual-Blockchained structure for contract-theoretic LoRa-based information systems. *Information Processing & Management*, *58*(3), Article 102492.

Zhao, Quanyu, et al. (2020). Blockchain-based privacy-preserving remote data integrity checking scheme for IoT information systems. *Information Processing & Management*, *57*(6), Article 102355.