EVMFuzz: Differential fuzz testing of Ethereum virtual machine

Ying Fu | Meng Ren | Fuchen Ma | Xin Yang | Heyuan Shi | Shanshan Li | Xiangke Liao

1School of Computer, National University of Defense Technology, Hunan, China
2School of Software, Tsinghua University, Beijing, China
3Big Data Institute, Central South University, Hunan, China

Correspondence
Shanshan Li, School of Computer, National University of Defense Technology, Hunan, China.
Email: shanshanli@nudt.edu.cn

Funding information
National Key Research and Development Program of China, Grant/Award Numbers: 2022YFB3104000, 2022YFB3104003; National Natural Science Foundation of China, Grant/Award Numbers: 61872373, 62272473, 62202500, 62177046; Hunan Natural Science Foundation, Grant/Award Number: 2021J30866

Abstract
The vulnerabilities in Ethereum virtual machine (EVM) may lead to serious problems for the Ethereum ecosystem. With lots of techniques being developed for the validation of smart contracts, the testing of EVM has not been well-studied. In this paper, we propose EVMFuzz, the first that uses the differential fuzzing technique to detect vulnerabilities in EVM. The core idea of EVMFuzz is to continuously generate seed contracts for different EVMs' execution, so as to find as many inconsistencies among execution results as possible, and eventually discover vulnerabilities with output cross-referencing. First, we present the evaluation metric for the internal inconsistency indicator. Then, we construct seed contracts via predefined mutators and employ a dynamic priority scheduling algorithm to guide seed contract selection and maximize the inconsistency. Finally, we leverage different EVMs as cross-referencing oracles avoiding manual checking. For evaluation, we selected four widely used EVMs for the test, conducted large-scale mutation on 36,295 real-world smart contracts, and generated 253,153 smart contracts as initial seeds. Accompanied by manual root cause analysis, we found five previously unknown security bugs and all had been included in the common vulnerabilities and exposures (CVE) database.

KEYWORDS
differential testing, domain-specific mutation, EVM, fuzzing

1 INTRODUCTION

Ethereum is an open-source, public, blockchain-based distributed computing platform. It can be viewed as a transaction-based state machine, whose foundation is transaction execution. There are about 500,000 transactions running on Ethereum every day, most of which involve the execution of smart contracts. Over the past few years, the safety and security problems of blockchain transactions have emerged endlessly. In June 2016, the Blockchain industry’s largest crowdfunding project—TheDAO, was attacked due to a serious flaw in a smart contract function, resulting in more than three million Ether loss. Then, many efforts are devoted to safeguarding transaction security by ensuring rigorous code logic of smart contracts.

Ethereum Virtual Machine (EVM) is the heart of Ethereum, which is often called the operating system of the Ethereum technology and is responsible for the execution and maintenance of smart contracts. As the authentic platform and standard for executing smart contracts, if there are some vulnerabilities in EVM’s internal implementation, it will definitely lead to serious consequences such as Ethereum operation errors or transaction failures. Only contract validation cannot ensure the correctness of Ethereum transactions execution, it is of great urgency to secure EVM. To perform efficient EVM testing, we need to solve the following challenges:
How to define general scenarios and evaluation metrics? EVM has at least 10 widely used implementations of different programming languages, all of which are based on the standards of Ethereum Yellow Paper. For example, the amount of EVM code in Geth platform is about 5475 lines of Go, accounting for 27.6% of the core infrastructure implementation of Geth. The implementation of EVM is complex, involving a large number of control and storage structures. It is vital to define meaningful and general evaluation metrics for the testing of different EVMs due to the diversity of EVM version and code complexity.

How to generate test cases that trigger EVM bugs? The Ethereum Yellow Paper provides the basis for EVM implementations, in which the functions and attributes are defined by formulas and rules. But there are neither authoritative test suites and benchmarks nor mature testing tools for EVM. Due to the lack of official benchmarks and widely used EVM testing frameworks, we urgently need to find out a solution that is able to perform efficient and accurate EVM testing and automatically generate EVM inputs fast and effectively.

To address these challenges, we propose EVMFuzz, which aims to automatically generate adversarial test inputs for EVMs. First, we define a general evaluation metric for the differential fuzzing of EVMs. As most EVMs are implemented as a transaction-based state machine, and the change of state depends on the opcode sequence to be executed, the input parameters and the gas limit, hence, we use the opcode sequence executed and gasUsed as two important indicators to evaluate EVMs’ performance on each test contract. EVMFuzz integrates different EVMs and creates a unified running environment for them. In this way, it takes the natural advantages of EVM’s multiple versions to quickly discover the output inconsistencies without manual checking. Then, our seed contract mutation and selection algorithms can continuously generate contracts that enlarge the metric difference, so that EVMFuzz can efficiently mine cases that trigger differential performance of EVMs and try to get those corner cases with inconsistent execution output. For evaluation, we chose four widely used EVM implementations, conducted empirical studies on 36,295 real-world smart contracts from Etherscan. Through guided fuzzing, 1596 variant contracts successfully triggered inconsistent execution output among different EVMs. With manual analysis, we found five previously unknown security bugs in those EVMs, and all had been included in the common vulnerabilities and exposures (CVE) database.

We make the following main contributions:

- We introduce an evaluation metric for EVM differential fuzzing, define eight mutators for seed contract generation and design a dynamic priority scheduling algorithm for seed contract selection.
- We implement EVMFuzz, the first differential fuzz testing framework, to efficiently detect the differences and vulnerabilities of different EVMs.
- We apply EVMFuzz to test some most widely used EVM versions, many inconsistencies and security bugs are detected, and five vulnerabilities have been assigned with unique CVE IDs.

The rest of this paper is organized as follows. We first introduce a motivating example in Section 2. We survey related work in Section 3. We provide the overview the of details of EVMFuzz in Section 4 and Section 5, respectively. The evaluation results and discussion are shown in Section 6 and Section 7. Finally, we conclude this paper in Section 8.

2 MOTIVATING EXAMPLE

To investigate the effectiveness of EVMFuzz, we use a simple example presented below. forTest is a simple contract with a function Testwhile, whose data structure is a while loop. When the input parameter \(a\) is less than \(b\), variable \(x\) will continually increase. Strictly speaking, the implementation of Testwhile function has a serious problem and may not appear in real life, if the input parameter satisfies the loop condition, it will result in an infinite loop. However, such contract can pass the check of most existing contract testing and verification tools, and our experiment proves that it can trigger different behaviors of multiple EVM implementations and even cause the denial of service problem.

```solidity
pragma solidity 0.4.24;
contract forTest {
  function TestWhile (uint a, uint b) public {
    uint x = 1;
    while (a < b) {
      x += 1;
    }
  }
}
```

Listing 1: The source code of smart contract named forTest.

We test the contract on three different EVM versions—JavaScript implementation js-evm, Python implementation Py-EVM, and C++ implementation aleth. When the TestWhile function is called with parameters \(a = 1, b = 2\), js-evm will continually print trace information, together with the increase of CPU usage and terminate the operation correctly; Py-EVM will print trace information and terminate the operation correctly;
aleth will not print trace information, but continue to occupy CPU resources until the system kills the whole process and crashes. Comparing the execution results, we find that aleth can neither handle the extreme situation of infinite loop properly that has no friendly user interaction or information feedback nor cut the loss in time and reduce malicious resource occupied, eventually leading to potential denial of service. Besides, We have applied for a CVE Id: CVE-2019-7710 for this problem.

This example shows that some contracts containing corner cases can trigger the boundary condition of EVM implementations and expose unexplored defects. But these contracts involving extreme circumstances are often inconsistent with logic programming rules, which require artificial construction, in other words, contract mutation. Furthermore, for massive mutated contracts, test oracle is difficult to define, and some extreme cases are not designated in EVM design specifications. Therefore, it is an efficient and effective way to apply differential fuzzing.

3 | RELATED WORK

3.1 | Fuzzing technique

Fuzzing is an automatic testing technique that covers numerous boundary cases using invalid data as application input to better ensure the absence of exploitable vulnerabilities. It is between manual testing and automated testing, and solves the problems of the strong dependence on testers’ professional level and the mismatch of test suites. Although the idea and operation of fuzzing is simple, it is still a fast, effective and low-cost testing technology.

Some popular AFL family tools apply various strategies to boost fuzzing process, including symbolic execution, schedule algorithm and so on. For example, EnFuzz integrates multiple fuzzing strategies to obtain better performance and generalization ability than that of any constituent fuzzer alone. There are also some tools focus on fuzzing in other domains, for example, Tardis and Rtkaller apply fuzzing technology to embedded system. Griffin aims to DBMS fuzzing. QuanFuzz is a search-based test input generator for the quantum programs.

3.2 | Differential testing

Differential testing is a popular software testing technology. Its core idea is to provide the same input to a series of similar programs and detect errors by observing their difference in execution results. Differential testing requires at least two reference objects and a set of standard test cases. If a program performs differently during the process or occurs a crash and other abnormal conditions, then it may have internal errors or defects.

Differential testing has been very successful in uncovering differences between independent implementations with similar intended functionality. For example, Chen et al.’s perform differential testing of JVMs using MCMC sampling for input generation. Differential testing has also been used for deep learning systems. DeepXplore was presented as the state-of-the-art white-box differential testing framework for deep learning systems and first introduce the concept of neuron coverage as a testing metric. DLFuzz extends the differential testing framework for DL systems with the comparisons of multiple similar inputs and does not need multiple platforms.

3.3 | Smart contract validation

Smart contract is executable code stored in the blockchain, that allows users to conduct trusted transactions without the participation of third-party organization. However, smart contracts have been shown to be exposed to severe vulnerabilities, and then many efforts have been devoted to ensuring smart contracts’ correctness. Some researchers use static analysis, like Oyente and Zeus. Using static analysis can quickly analyze the code logic, but it also has a high false-positive rate. So dynamic analysis is also used in the field of smart contract validation, like V-Gas and ContractFuzzer. The authors of another tool SCStudio also combine widely-used methods for finding smart contract vulnerabilities in order to secure smart contracts with minimal false positives and negatives.

3.4 | Main difference

Different from the above work, EVMFuzz mainly focuses on discovering the defects and vulnerabilities in the most significant Ethereum infrastructure, EVM. Due to the distinctiveness of EVM input and the diversity of EVM implementations, traditional testing techniques cannot be directly applied to EVM testing. EVMFuzz extends our premier work EVMFuzzer with more mutate strategies and seed selection algorithms on smart contracts, and hence, is more efficient to detect those EVM vulnerabilities. EVMFuzz is the first that uses differential fuzz testing technique to detect vulnerabilities of EVM. Particularly, EVMFuzz takes advantage of EVMs’ multi-implementation to quickly find output discrepancies and
reduce manual checks. Within EVMFuzz, we also define the domain-specific EVM test indicators to guide the differential fuzzing process with different contract mutation and selection strategies.

4 | APPROACH OVERVIEW

In this section, we briefly introduce the workflow of EVMFuzz. Our goal is to apply differential fuzz testing on EVMs. The concept of differential fuzz testing is very simple, that is, to continuously provide invalid, unexpected or random data as inputs to several programs with the same functions. These programs are then monitored for catching “different act” on some inputs, if so, we may find a bug in some of the programs. In this paper, our test objects are the same functional EVM platforms implemented by different programming languages, and the test input is the mutated smart contract. An overview of EVMFuzz is given in Figure 1, which consists of two major components, that is, seed contract generation based on static analysis and unified EVM execution module. We will also introduce the evaluation metrics for EVM differential fuzzing.

4.1 | Seed contract generation

The input for the seed generation module is the smart contract file, and the output is a contract variant whose key property has been modified by specific mutators. First, we precisely construct the Critical locations identified Abstract Syntax Tree (CAST) of the seed contract (Section 5.1), for facilitating subsequent mutation and analysis. Then the seed contract will be put into the seed pool. EVMFuzz will rank the candidate contracts as a prioritized queue under the guidance of dynamic priority, and the contract in the first place will be selected as the next subject (Section 5.2). After choosing the contract for mutation, EVMFuzz uses eight predefined mutators and the combination strategy to guide mutation (Section 5.3), and obtains the input of unified EVM execution module. The goal is to generate contracts that can increase the degree of metric difference and trigger different execution outputs.

4.2 | Unified EVM execution

EVM execution module provides a unified runtime environment for various EVMs (Section 5.4). After receiving the contract file from the seed generation module, it compiles the seed into EVM bytecode. The input parameter is generated according to the data type of the called function, thus, the uniform input for each EVM is obtained. Then EVMFuzz automatically runs all EVMs, calculates the difference information according to the test metric, and compares the execution output results. Finally, according to the seed’s ability to enhance the degree of metric difference, EVMFuzz decides whether to put the seed contract into the seed pool where high-quality seeds are preserved (Section 5.5). Besides, when the execution output is inconsistent, this module will also record the potential exception for manual root cause analysis.

4.3 | Metrics formulation

To evaluate the performance of each EVM on the test contract, we define the metric on the basis of two general indicators. EVMs are implemented as a transaction-based state machine, and the change of state depends on the sequence of opcode to be executed, the input parameter, and the gas limit. Hence, we use the internal opcode sequence executed and gasUsed as the two indicators.

1. **opcode sequence.** Opcode is short for “operation code”, which is used to describe the part of machine code that performs some sort of operation in machine language instructions. From the perspective of computer instruction execution, each function call is completed by a series of
opcode execution. The opcode sequence clearly shows a complete process of contract operation, which can be used to check the correctness of each step. For platform $i$, we define $\text{opSeqLen}(i,C)$ as the length of opcode sequence of $i$ when executing contract $C$.

2. \textbf{gasUsed}. Gas is a unit used to measure the workload of an operation in Ethereum. If an operation consumes more resources, it consumes more gas. This mechanism ensures that resources are not maliciously wasted. $\text{gasUsed}$ is the total gas unit cost of one transaction. Here we use $\text{gasUsed}(i,C)$ to represent total gas consuming of platform $i$ after running contract $C$.

Based on these two indicators, we further define the evaluation metric of difference information. When given an input parameter, the normal execution of a transaction on a dedicated EVM platform is determined by a confirmed and unique execution sequence, and the total gas consumption is also calculated. Therefore, we construct an evaluation metric $\text{diff}$ to measure the difference among different EVMs execution (Section 5.5). The greater the metric difference, the higher the probability of inconsistent execution output. Execution output is the return value after all executions, that $\text{output}(i,C)$ is defined as the returns of $C$'s execution on EVM $i$. For a function call, it is the returned data, and for a transaction, it is the balance. While the metric defined on the two internal indicators reflects the implementation and execution differences of different EVMs, the execution output can intuitively reflect whether those EVMs are running consistently or correctly.

5 | EVMFUZZ DESIGN

In this section, we will elaborate on the key components of EVMFuzz.

5.1 | CAST construction

```solidity
pragma solidity 0.4.24;
contract Demo {
    function transfer (address from, address caddress, address[] _tos, uint v) public returns (bool) {
        require(_tos.length > 0);
        bytes4 id=keccak256("transferFrom(address,address,uint256)"));
        for(uint i=0; i<_tos.length; i++) {
            caddress.call(id,from,_tos[i],v);
        }
        return true;
    }
}
```

![Figure 2](image-url) The source code and the CAST structure of smart contract named Demo, the shaded nodes are the identified critical locations.
Before EVMFuzz starts the entire procedure of fuzzing, it first carries out static analysis on initial seed contracts and generates the Critical locations identified Abstract Syntax Tree (CAST) structure for further mutation.

A CAST of a smart contract is a structured tree representation of the abstract syntactic structure of Solidity source code. Each node of the tree denotes a construct occurring in the source code. CAST can define and decompose properties in all statements of contract. Transforming a contract into CAST structure can help us complete the subsequent contract mutation operations. It can directly search, replace, delete or insert operators according to the key attributes. Furthermore, CAST identifies critical locations of a seed contract, which are the subtree of statements related to ether transfer. It mainly involves six statement symbols—`new`, `call`, `delegatecall`, `callcode`, `send` and `transfer`.

Formally, a CAST is a tuple \((N, \text{root}, E)\) such that,

- \(N\) is a set of nodes. Each node represents the basic structure in the Solidity source code, and \(N.\text{critical}\) represents whether the node belongs to a critical location.
- \(\text{root} \in N\) is the CAST root node, generally the contract name.
- \(E \subseteq N \times N\) is a set of edges. An edge \((n, n')\) is in \(E\) if and only if there is a semantic relationship between node \(n\) and node \(n'\).

Based on CAST, we can guide pre-defined mutators to select the structures that are identified as critical locations in order to test the core functions of EVM.

A simple example is presented in Figure 2, where the shaded nodes are regarded as the critical locations, for the reason that they are all under the subtree of the `call` statement.

### 5.2 Seed contract prioritization

In seed contract pool, the importance of each candidate contract is different. In general, the contract that makes the metric difference among EVMs larger should be the benchmark for the next mutation iteration. But at the same time, in order to ensure the diversity, other contracts should also have a certain probability of being selected. Therefore, we use the dynamic priority scheduling algorithm to maintain a candidate queue. For each contract, we give it an initial priority, and then its value changes with the increasing of waiting time to ensure that every seed has the chance to be selected.

As Algorithm 1 shows, the priority of each seed contract consists of two parts (Algorithm 1 line 3-4). The first part is metric difference priority, and the initial value is a number between 0 and 10, which is proportional to the value of difference; the second part is time priority, and the initial value is 0. Then, all candidate seed contracts are sorted according to the priority value, and the contract with the highest integrated priority is selected as the next mutation object, and the time priority of other seed contracts is increased for the next iteration (Algorithm 1, lines 7–9).

**Algorithm 1: Seed prioritization**

**Input:**
- List of candidate contracts, `candidate_seeds`
- Difference priority of each seed contract, `diff_pri`
- Time priority of each seed contract, `time_pri`

**Output:**
- Test contract for the next iteration, \(C\)

```plaintext
1 priority = [/* Store the priority of each contract */]
2 for i = 0 to len(candidate_seeds) do
3     cur = candidate_seeds[i]
4     priority[cur] = diff_pri[cur] + time_pri[cur]
5 end
6 candidate_seeds.sort(cmp=priority, reverse=True)
7 for i = 1 to len(candidate_seeds) do
8     time_pri[candidate_seeds[i]] += 1
9 end
10 C = candidate_seeds[0]
11 return C
```
5.3 | Seed contract mutation

The goal of EVMFuzz is to generate high-quality seed contracts that can trigger more discrepancies among EVMs and to identify vulnerabilities in EVMs. As Algorithm 2 shows, EVMFuzz first generates CAST from contract’s Solidity code (Algorithm 2, line 1). Then, it updates the weight of each mutator based on the metric difference feedback (Algorithm 2, line 2), and selects a combination strategy to mutate the candidate seed contract with the corresponding mutators (Algorithm 2, lines 3–6). Finally, EVMFuzz reconstructs the code from CAST (Algorithm 2, line 7). Details about the mutators and mutation strategies are introduced as below.

### Algorithm 2: Contract mutation

**Input:**
- Contract before mutation, $C$
- Difference information of last time iteration, $\text{diff}$
- Selection of combination strategy, $\text{choice}$
- List of mutators, $\text{mutator}$

**Output:**
- Contract after mutation, $C'$

1. $C, \text{CAST} \leftarrow$ generate the CAST of $C$
2. $\text{mutator}.\text{sort}(\text{cmp}=\text{diff})$
3. $m \leftarrow$ selected mutators from $\text{mutator}$ based on $\text{choice}$
4. for $i = 1$ to $\text{len}(m)$ do
   5. $C,\text{CAST}'' \leftarrow$ mutate the $C,\text{CAST}$ using $m[i]$
6. end
7. $C' \leftarrow$ source code generated based on $C,\text{CAST}''$
8. return $C'$

### 5.3.1 | Typical mutators

During mutation, we must ensure the syntax correctness of the modified contract so that it can generate bytecode normally. Due to the need to ensure the syntax correctness of the mutated contracts, we did continuous definition of the mutation strategy and experimental verification, and finally we selected eight mutators according to the functional logic features of the smart contract, as shown in Table 1. These variations are based on three different granularities: the first is word-level (rows 2–3), modifying variable and function attributes, which could influence the storage and syntax structure; the second is character-level (rows 4–6), modifying the arithmetic or conditional operators and the terminate condition in a loop, which may stochastically change the control flow; and the last is statement-level (rows 7–9), like inserting or deleting some assert statements that are used for internal condition judgment.

### 5.3.2 | Mutator selection

Each mutator performs differently, so we can also maintain a priority queue based on the feedback metric difference. For a seed contract, we update the weight of corresponding mutators after the multi-version EVMs comparison, which is similar to the initialization of metric difference priority (Section 5.2). If the metric difference increases, the mutator ID is pushed into the queue in descending order according to the weight; otherwise, the queue will not update. Except for the weight update, we design five mutator combination strategies to further increase the randomness and diversity of mutation in each iteration. Let $\text{Mutators}$ denote the queue of mutators which are sort by weight from highest to lowest, $\text{Mutators} = [\text{mu}_1, \text{mu}_2, \ldots, \text{mu}_n]$. Let $\text{Comb}$ denote the array of mutator combination strategies, $\text{Comb} = [\text{Comb}_1, \text{mu}_2, \ldots, \text{mu}_5]$. The detailed definition is shown below:

- $\text{Comb}_1$: OddComb, combination of the mutators whose index is odd.

  $$\text{Comb}_1 = \text{OddComb} = \{ \text{mu}_i | i = 2k - 1, k \in \mathbb{N}^+, i < n \}$$

- $\text{Comb}_2$: EvenComb, combination of the mutators whose index is even.

  $$\text{Comb}_2 = \text{EvenComb} = \{ \text{mu}_i | i = 2k, k \in \mathbb{N}^+, i < n \}$$
Combi: ExtremeComb, combination of first and last mutator.

\[ \text{Combi} = \text{ExtremeComb} = \{ \mu_i | i = 0, i = n \} \]

Comby: RandomComb, randomly select a mutator, without weight.

\[ \text{Comby} = \text{RandomComb} = \{ \mu_i | i = \text{Random}(1, n) \} \]

Combiz: AllComb, randomly choose one strategy above in each iteration.

\[ \text{Combiz} = \text{AllComb} = \text{Combi} | i = \text{Random}(1, 5) \]

### 5.4 Unified EVM execution

After an effective contract mutation, it is necessary to compare the metric difference of the EVMs’ execution in order to guide the subsequent mutation and selection of seed contracts. The execution of different EVMs requires unified management. The EVM execution module in Figure 1 is responsible for the entire execution.
The first step is to get the data that can directly feed into EVM platform, namely, the contract bytecode and input parameters. By executing `solc -bin -runtimeexec.sol`, we can compile the mutated seed contract `C` into runtime bytecode. According to the selected function's received data type, we can generate the input parameters. For each data type, we pre-define some common or extreme values, which are randomly selected at the time of generation. The second step is to call the execution interface of each EVM to execute the contract data, standardize the output format of the execution result and save the output. The third step is to analyze the output files of each platform, compare the data for each indicator and calculate the metric difference for the next iteration.

5.5 | Seed contract selection

As mentioned above, we evaluate the quality of a generated seed contract based on platform diversity, so we select candidate seed `C` based on the metric difference `diff`, which is defined as follows:

\[
gasDiff(i, j) = \text{norm}(\text{abs}(\text{gasUsed}(i, C), \text{gasUsed}(j, C)))
\]

\[
opDiff(i, j) = \text{norm}(\text{abs}(\text{opSeqLen}(i, C), \text{opSeqLen}(j, C)))
\]

\[
diff = gasDiff(i, j) + \text{opDiff}(i, j)
\]

\[
\text{outVul}(i, j) = \text{output}(i, C) \oplus \text{output}(j, C)
\]

Difference is used to measure the input seed's ability of inducing platforms to make differential decisions. If a mutated contract enlarges the difference after executing on different EVMs, we consider it to be a high-quality seed with a higher probability of triggering vulnerabilities and store it in the seed queue as a candidate for the next iteration of prioritization and mutation.

6 | EVALUATION

In this section, we present the experiment details. EVMFuzz has conducted large-scale mutations on 36,295 real-world smart contracts and several EVM discrepancies and vulnerabilities have been found. We answer the following three questions: (i) Is there any inconsistency among EVMs? (ii) Could EVMFuzz generate high-quality seed contracts efficiently? (iii) Is it possible to find EVM bugs by differential fuzz testing?

6.1 | Data and environment setup

All experiments were performed atop a machine with eight cores (Intel i7-7700HQ @3.6GHz), 16GB of memory, and Ubuntu 16.04.4 as the host operating system. The 36,295 real-world contracts we used for mutation were crawled from the Etherscan. EVMFuzz tested four widely used EVM platforms for each mutated contract, namely, js-evm (v2.4.0), Py-EVM (v0.2.0-alpha.33), aleth (v1.5.0-alpha.6), and geth (v1.8.17).

All those initial real-world contracts are deployed in Ethereum. We first analyzed them in four aspects: lines of code, functions, all relevant opcodes and some specific opcodes. More than 6500 contracts have over 500 lines of code, and 90 of them have more than 2000 lines of code, which shows that the target contracts we choose are complicated. Because our experiment is based on the function level, we find that the total function number runs up to 1,013,013 and 83% contracts have more than 15 inner functions, which provides a wide range of space for mutation.

In essence, EVM executes opcode sequence rather than the source code written in high-level languages, whose completeness is closely related to the ability to handle distinct opcodes. The number of instructions in a single contract ranges from 32 to 83,329, with an average of 6472. In order to ensure the normal execution of all transactions, each EVM must correctly handle the logic of 140 different instructions. The statistics on the occurrences of 11 representative opcodes in each contract are analyzed. The opcodes that are related to stack (like PUSH1, POP, and DUP1) or conditional jumps (like AND, ISZERO, and JUMPDEST) have a higher frequency of occurrence, which has also been considered in the mutator's design.

6.2 | Is there any inconsistency among EVMs?

From the most intuitive statistics, among those real-world contracts, 34,699 contracts were successfully executed by four platforms and the execution outputs were the same. Based on these initial contracts, we will elaborate on the metric difference among different EVMs in the view of the two internal test indicators: `gasUsed` and opcode sequence.
6.2.1 | gasUsed inconsistency

Within the 34,699 successfully executed contracts, there are 33,424 contracts that are executed normally with the same opcode sequence. They are used for gasUsed comparison, which excludes the inconsistency of gas consumption caused by different execution opcode sequences.

In the process of contract execution, almost every platform has over 50% average inconsistency rate of gasUsed with others, and aleth even produces a different gas consumption over 90% of contracts. There are two reasons for this phenomenon. For one reason, the gasCost for some of the same opcodes defined by different EVM platforms is inconsistent. But for the reason that all EVMs are implemented based on the Ethereum Yellow Paper, this situation rarely occurs. For another, the Ethereum Yellow Paper did not specify specific gas consumption values for some opcodes, such as MSTORE, SLOAD, JUMPDEST. Then, different platforms adopt different refund mechanisms, which also results in inconsistency.

For further analysis, we calculated the average gasUsed of these 33,424 smart contracts as the baseline, then the gas consumption of each EVM is compared with it, as presented in Figure 3A–D. The y-axis of Figure 3A–D represents gasUsed, and the x-axis represents these 33,424 smart contracts. From Figure 3, we can see that js-evm consumed less gas than the baseline, while Py-EVM consumed more, geth and aleth consumed almost the same as the average. Based on these observations, developers should pay attention and set different gasLimit accordingly when deploying the same contract on different EVMs to ensure the correct transaction.

6.2.2 | Opcode sequence inconsistency

When analyzing the experimental results, we found that even though the execution outputs of the four EVMs were the same, their opcode sequences were not completely consistent. In total, 1275 seed contracts were successfully executed on four EVM platforms and returned the same output, but the length of opcode sequence generated by executing contract transactions on those EVM platforms are not exactly the same, results are shown in Table 2. From Table 2, we can see that the opcode sequence length of these 1275 contracts was always the same on geth and aleth. However, on js-evm and Py-EVM, the opcode sequence length was different from others. The inconsistency in sequence length may be due to the fact that each EVM platform internally optimizes when executing the bytecode.

From the above statistics, it is reasonable to conclude that there are inconsistencies among the implementation and execution of different EVMs, and it is possible to leverage the metric difference of gasUsed and opcode sequence indicator to guide the generation of contracts resulting in potential inconsistent execution output.

6.3 | Could EVMFuzz generate high-quality seeds efficiently?

Within three days, based on the crawled real-world initial contracts, EVMFuzz generated and executed 253,153 non-redundant contracts. Among them, more than half of the seed contracts (66.2%) successfully trigger the differential performance of gasUsed and opcode sequence indicator,
including 1596 variant contracts showed inconsistent output results among the four EVM platforms. In the process of mutation, the generated seed contracts may contain some rare situations, such as illegal data types or infinite loops, due to the change of key attribute values. If an EVM platform cannot deal with these boundary conditions properly, it will not get the correct output result, and even lead to some serious security problems, as the motivating example presented in the Section 2.

To evaluate the efficiency of different mutation strategies and the quality of each mutator, we selected a contract with the longest LOC from the crawled real-world contracts for detailed demonstration. This function receives two parameters and returns a uint type, including two loops and six branch structures. Each branch contains a different number of conditional judgments, arithmetic operations and assert statements, which provide a large mutation space.

As mentioned in Section 5.3, we designed five strategies for mutator combination, and their performance statistics are presented in Figure 4. At the beginning of mutation, the EvenComb strategy had the fastest rising rate. Within 30 iterations, the metric difference diff increased from 346 to 1840. However, it then had difficulty jumping out of the local optimal solution and grew slowly. After the 80th iteration, AllComb strategy found a new breakthrough and began to grow rapidly. Eventually, it had the largest metric difference. It also shows that the random choice of different strategies with guiding information in each iteration can generate high-quality seeds most efficiently. As for the RandomComb strategy without weight guiding information, which is completely based on randomness, it had the worst performance in terms of rising rate and final value. Those statistics indicate that the weight information of each mutator has a certain guiding significance and facilitates the combination mutation strategy.

To investigate the effects of each mutator and the corresponding weight, we analyze the AllComb strategy in detail. Firstly, we generate a tuple of input parameters based on the function signature. Then, with the function signature and input parameters unchanged, different strategies are randomly adopted to mutate the original contract for 50 iterations. After that, the feedback metric difference and the corresponding mutator weights in each iteration are recorded, shown in Figure 5. The mutator ID matches the first column of Table 3, which shows the mutator weight statistics after mutating 36,295 contracts. There are eight mutators involved, with an average weight of 0.125, so we use this average value as a baseline to observe whether each mutator plays a positive or negative role in each iteration of the AllComb strategy.

In Figure 5B, C, most regions are concentrated above the baseline, and the peak value even reaches 0.6. These two mutators are respectively used for arithmetic operators and conditional operators modification. For the arithmetic operators, the mutator changes the results of computation and affects the subsequent execution path; and for the conditional operators, the mutator changes the executed module, involves different execution opcodes and causes the increasing of difference. On the contrary, in Figure 5G, most areas are concentrated below the baseline, indicating that this mutator didn’t play a leading role in most iterations. The reason is that deleting the return value of a function is an irreversible operator and will not have effect after one modification. The remaining five mutators’ weights are mainly near the baseline (±0.05). Among them, #1 and #5 have a higher peak value, which shows that they are easier to trigger the trend of rising under some corner cases; meanwhile, #4, #6 and #8 have little influence on the increasing of difference.

From the statistics, it is reasonable to conclude that EVMFuzz could efficiently generate corner cases to trigger different outputs among EVMs with guiding information of gasUsed and opcode sequence.

### 6.4 Is it possible to find EVM bugs by differential fuzz testing?

We have demonstrated that EVMFuzz can efficiently generate seed contracts and perform differential fuzz testing to find smart contract execution differences across different EVM platforms. After discovering thousands of output inconsistencies, we conducted the manual analysis and tried to explore the root causes.

![Figure 4](image-url)  
**Figure 4.** Efficiency of different mutation strategies.
We ensured its reproducibility and then carefully reviewed the source code of EVMs. Finally, we found defects in those EVM platforms, of which five vulnerabilities were registered as Common Vulnerabilities and Exposures, numbered as CVE-2018-18920, CVE-2018-19183, CVE-2018-19184, CVE-2018-19330, and CVE-2019-7710. Table 4 lists the relevant information for these CVEs.

Except for these five EVM vulnerabilities, there were only six existing vulnerabilities associated with EVM among 112,913 CVE entries, whereas smart contract vulnerabilities were over 500. We select one previous unknown CVE for a detailed explanation.

**CVE-2018-18920**: It is a runtime error on Py-EVM v0.2.0-alpha.33. Users can use function `vm.execute_bytecode` in Py-EVM to execute the smart contract bytecode. Function `vm.execute_bytecode` calls the function `vm.state.get_computation` to handle the opcode operation. But when the user sets the ether transfer mode to execute a contract that is not able to do the transfer operation (like the code snippet shown in Figure 6, it is a simple contract named Origin with one construct function.), the Py-EVM cannot handle it normally because of the improper handling of function `vm.state.get_computation`. The execution result is shown in Figure 6, computation error occurs. This issue could result in endless smart contract execution without gas being paid. Once this vulnerability is exploited, the attackers can make the Ethereum network abnormally congested without cost, which will cause the Ethereum ecology to be imbalanced.

We ensured its reproducibility and then carefully reviewed the source code of EVMs. Finally, we found defects in those EVM platforms, of which five vulnerabilities were registered as Common Vulnerabilities and Exposures, numbered as CVE-2018-18920, CVE-2018-19183, CVE-2018-19184, CVE-2018-19330, and CVE-2019-7710. Table 4 lists the relevant information for these CVEs.

Except for these five EVM vulnerabilities, there were only six existing vulnerabilities associated with EVM among 112,913 CVE entries, whereas smart contract vulnerabilities were over 500. We select one previous unknown CVE for a detailed explanation.

**CVE-2018-18920**: It is a runtime error on Py-EVM v0.2.0-alpha.33. Users can use function `vm.execute_bytecode` in Py-EVM to execute the smart contract bytecode. Function `vm.execute_bytecode` calls the function `vm.state.get_computation` to handle the opcode operation. But when the user sets the ether transfer mode to execute a contract that is not able to do the transfer operation (like the code snippet shown in Figure 6, it is a simple contract named Origin with one construct function.), the Py-EVM cannot handle it normally because of the improper handling of function `vm.state.get_computation`. The execution result is shown in Figure 6, computation error occurs. This issue could result in endless smart contract execution without gas being paid. Once this vulnerability is exploited, the attackers can make the Ethereum network abnormally congested without cost, which will cause the Ethereum ecology to be imbalanced.

```solidity
pragma solidity 0.4.24;
contract Origin {
    address public owner;
    /*
     * @dev The Ownable constructor sets the original owner
     * of the contract to the sender contract.
     */
    function Origin() {
        owner = msg.sender;
    }
}```
We propose a differential fuzz testing framework and have found some bugs in EVM, but there are still some deficiencies that need to be improved in our future work.

1. **More mutators design.** Considering the correctness of compiling and the logical complexity, we designed eight mutators, whose quantity still needs improving. Currently, we just modify the key attribute values based on CAST. In the future, we will make more semantic level mutations based on source code, such as adding internal function calls or contract inheritance relations. In the future, we can even automatically generate contracts for testing, similar to the C program generation tool Csmith and the SQL generation tool SQLsmith.

2. **More efficient priority scheduling.** In order to maintain the priority queue of mutator and candidate seeds, we adopt the data structure of PriorityQueue, which is implemented by heap. The insertion and deletion operations take the complexity of $\Theta(\log n)$, and the time complexity of sorting is $\Theta(n\log n)$. Now, the scale of mutators is small and the length of candidate seeds is short, so the time overhead is small, but the storage structure still needs to be optimized to further improve efficiency. Furthermore, the mutation priority is now based on the sum of two parts, the first part is difference priority and the second part is time priority. We can introduce coefficients $a$ and $b$ to balance these two parts in the future. For example, the difference priority might account for more proportion than the time priority.

3. **Support for more EVM implementations.** The core idea of differential testing is testing on at least two objects with the same functions. Therefore, we analyze the differences in four EVM platforms: js-evm, Py-EVM, geth, and aleth. However, in the real world, many platforms based on other programming languages are also widely used, and they may also have some fatal vulnerabilities in implementation. We will further incorporate them into our platform.

### TABLE 4 Description of five high-risk vulnerabilities detected by EVMFuzz.

| CVE-ID       | Platform | Version       | Language | Description                                                                 | Created date        |
|--------------|----------|---------------|----------|-----------------------------------------------------------------------------|---------------------|
| CVE-2018-18920 | Py-EVM   | v0.2.0-alpha.33 | python   | Py-EVM v0.2.0-alpha.33 allows attackers to make a vm.execute_bytecode call that triggers illegal values shown in stack. | November 2, 2018    |
| CVE-2018-19183 | js-evm   | v2.4.0        | JavaScript | ethereumjs-vm 2.4.0 allows attackers to cause a denial of service (vm.runCode failure and REVERT) via a "code: Buffer.from(my_code, "hex")" attribute. | November 11, 2018   |
| CVE-2018-19184 | geth     | v1.8.17       | golang   | cmd/evm/runner.go in Go Ethereum (aka geth) 1.8.17 allows attackers to cause a denial of service (SEGV) via crafted bytecode. | November 11, 2018   |
| CVE-2018-19330 | aleth   | v1.5.0-alpha.6 | cpp ** RESERVED ** Details would be public after the vulnerability has been repaired to avoid potential attack. | November 17, 2018   |
| CVE-2019-7710  | aleth    | v1.5.0-alpha.7 | cpp ** RESERVED ** Details would be public after the vulnerability has been repaired to avoid potential attack. | February 10, 2019   |

**FIGURE 6** Invalid execution output of Py-EVM and the contract that can trigger the issue.

Invalid opcode execution crash
4. **More accurate selection metrics.** In the selection of mutated seed contracts, we use `gasUsed` and opcode sequence difference as the inducer for triggering EVM vulnerabilities. But there are still some problems, such as insufficient quantity and the lack of unity for data types, waiting for further improvement. Furthermore, even for the two difference indicators, we can also introduce coefficients \( a \) and \( b \) to balance the `gasUsed` and the opcode sequence difference in the future. For example, the weight of the opcode sequence difference might account more than that of `gasUsed` difference. We can even dynamically adjust the coefficients during the whole fuzzing stages.

8 | **CONCLUSION**

In this paper, we propose EVMFuzz, the first automated differential fuzz testing framework, to efficiently find vulnerabilities of EVMs implemented by different programming languages. We introduce the definition of EVM fuzz testing metrics: `gasUsed` and opcode sequence, which measure the internal difference in execution information between EVMs. We design eight mutators for smart contracts, so that EVMFuzz can generate plenty of seed contracts without syntax error in a short time. EVMFuzz uses the metric difference as guidance for seed preserving and employs a dynamic priority scheduling algorithm for selecting the contract in the next iteration. We evaluated EVMFuzz based on four widely used EVM implementations and conducted numerous mutations on 36,295 real-world smart contracts. Among the generated 253,153 smart contracts, more than half successfully showed the differential performance of `gasUsed` and opcode indicator, including 1596 variant contracts triggered inconsistent output results among the four EVM platforms. With manual root cause analysis, five vulnerabilities have been assigned unique CVE IDs. Our future work mainly includes developing more general smart contract mutators, generators and priority scheduling algorithms, and conducting more extensive evaluations on more EVMs.

**ACKNOWLEDGMENTS**

This work is supported by the National Key Research and Development Program of China (Nos. 2022YFB3104000 and 2022YFB3104003), National Natural Science Foundation of China (Nos. 61872373, 62272473, 62202500, and 62177046), and Hunan Natural Science Foundation (No. 2021JJ30866).

**DATA AVAILABILITY STATEMENT**

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

**ORCID**

Ying Fu https://orcid.org/0000-0001-5329-2586
Meng Ren https://orcid.org/0000-0003-2639-3941
Fuchen Ma https://orcid.org/0000-0002-1360-9814
Heyuan Shi https://orcid.org/0000-0002-9040-7247
Shanshan Li https://orcid.org/0000-0003-0798-974X
Xiangke Liao https://orcid.org/0000-0002-6125-3330

**ENDNOTE**

* EVMFuzz is an extended version of EVMfuzzer. More details in Section 3.4.

**REFERENCES**

1. Ethereum. ethereum.org. https://www.ethereum.org/; 2018.
2. Wood DG. Ethereum: A secure decentralised generalised transaction ledger, Vol. 151; 2014:1-32.
3. Etherscan. Ethereum transaction history. https://etherscan.io/chart/tv/; 2018.
4. Ethereum. Go ethereum. https://github.com/ethereum/go-ethereum; 2018.
5. Fu Y, Ren M, Ma F, et al. 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; 2019:1110-1114.
6. Etherscan. Ethereum (eth) blockchain explorer. https://etherscan.io/; 2018.
7. EthereumJS. The ethereum vm implemented in javascript. https://github.com/ethereumjs/ethereumjs-vm; 2018.
8. Ethereum. Python implementation of the evm. https://github.com/ethereum/py-evm; 2018.
9. Ethereum. C++ client, tools and libraries. https://github.com/ethereum/aleth; 2018.
10. Liang H, Pei X, Jia X, Shen W, Zhang JG. Fuzzing: State of the art. IEEE Trans Reliab. 2018;67:1199-1218.
11. Zalewski M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/; 2015.
12. Böhme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; 2016:1032-1043.
13. Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security; 2017:2329-2344.

14. Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; 2018:475-485.

15. Wang M, Liang J, Chen Y, et al. Safi: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion); 2018:61-64.

16. Liang J, Jiang Y, Chen Y, Wang M, Motyka O, Sun J-G. 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; 2018:809-814.

17. Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T. Kafl: Hardware-assisted feedback fuzzing for os kernels. In: USENIX Security Symposium; 2017:167-182.

18. Hertz, Newsham. Triforceafl - afl/qemu fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL/; 2015.

19. Zero GP. Winaff - a fork of afl fuzzing windows binaries. https://github.com/googleprojectzero/winafl. Accessed February 18, 2019; 2015.

20. Fu J, Liang J, Wu Z, Wang M, Jiang Y. Griffin: Grammar-free dbms fuzzing. In: 37th IEEE/ACM International Conference on Automated Software Engineering; 2022:1-12.

21. Pei K, Cao Y, Yang J, Jana S. Deepxplore: Automated whitebox testing of deep learning systems. In: proceedings of the 26th Symposium on Operating Systems Principles; 2017:1-18.

22. Guo J, Jiang Y, Zhao Y, Chen Q, Sun J-G. Dlfuzz: Differential fuzz testing of deep learning systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2018:739-743.

23. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive. 2016;2016:1007.

24. Hiri Y. Formal verification of deed contract in ethereum name service. 2016. November-2016.[Online]. Available: https://yoichihirai.com/deed.pdf

25. Luu L, Chu D-H, Olickel H, Saxena P, Hober A. Making smart contracts smarter. IACR Cryptology ePrint Archive. 2016;2016:633.

26. Kalra S, Goel S, Dhawan M, Sharma S. Zeus: Analyzing safety of smart contracts. In: Nds; 2018:1-12.

27. Ren M, Ma F, Yin Z, et al. Scstudio: A secure and efficient integrated development environment for smart contracts. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2021:666-669.

28. Kersten ML, Koutsourakis P, Manegold S, Zhang Y. Squalpel: A database performance platform. 1-10; 2011.