EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing

Ying Fu
KLIISS, BNRist, Tsinghua University
China

Meng Ren
KLIISS, BNRist, Tsinghua University
China

Fuchen Ma
KLIISS, BNRist, Tsinghua University
China

Heyuan Shi
KLIISS, BNRist, Tsinghua University
China

Xin Yang
KLIISS, BNRist, Tsinghua University
China

Yu Jiang
KLIISS, BNRist, Tsinghua University
China

Huizhong Li
WeBank
China

Xiang Shi
WeBank
China

ABSTRACT

Ethereum Virtual Machine (EVM) is the run-time environment for smart contracts and its vulnerabilities may lead to serious problems to the Ethereum ecology. With lots of techniques being continuously developed for the validation of smart contracts, the testing of EVM remains challenging because of the special test input format and the absence of oracles. In this paper, we propose EVMFuzzer, the first tool that uses differential fuzzing technique to detect vulnerabilities of EVM. The core idea is to continuously generate seed contracts and feed them to the target EVM and the benchmark EVMs, so as to find as many inconsistencies among execution results as possible, eventually discover vulnerabilities with output cross-referencing. Given a target EVM and its APIs, EVMFuzzer generates seed contracts via a set of predefined mutators, and then employs dynamic priority scheduling algorithm to guide seed contracts selection and maximize the inconsistency. Finally, EVMFuzzer leverages benchmark EVMs as cross-referencing oracles to avoid manual checking. With EVMFuzzer, we have found several previously unknown security bugs in four widely used EVMs, and 5 of which had been included in Common Vulnerabilities and Exposures (CVE) IDs in U.S. National Vulnerability Database.

The video is presented at https://youtu.be/9Lejgf2GSOk.

1 INTRODUCTION

Ethereum can be viewed as a transaction-based state machine [33]. Ethereum Virtual Machine (EVM) is often called the operating system of the Ethereum technology and is responsible for the execution and maintenance of smart contracts. Over the past few years, the safety and security problems of the transactions have emerged endlessly, causing huge property loss. As the authentic platform and standard for Ethereum transaction executing, if there are vulnerabilities in EVM’s implementation, it will definitely lead to serious consequences. At present, EVM has at least 10 widely used implementations of different programming languages [7]. Lack of mature testing tool for EVM, it is difficult to guarantee the security of EVM. So, it is of great urgency to find an efficient way to secure EVM.

In this paper, we present EVMFuzzer, the first automated differential fuzz testing tool to efficiently mine vulnerabilities of EVMs implementations. EVMFuzzer firstly defines and uses the opcode sequence executed and gas used as two important indicators to evaluate EVMs’ execution differences on each test contract. It integrates some of the widely used EVMs as benchmarks and creates a unified running environment for the target EVM and benchmark EVMs. In this way, it takes the natural advantages of differential testing to quickly discover the output inconsistencies without manual checking. Then, the seed contract generation module can continuously generate contracts that enlarge the metric difference, so that EVMFuzzer can efficiently mine cases that trigger differential performance of EVMs and try to get those corner cases with inconsistent execution output.

For evaluation, we collected 36,295 real-world smart contracts from Etherscan [11] as our initial seeds. Through guided fuzzing, 1,596 variants of those initial seed contracts triggered inconsistent execution output among different EVMs. With manual root cause analysis, we found several previously unknown security bugs in four widely used EVMs, and 5 of which had been included in Common Vulnerabilities and Exposures (CVE) database [24].
2 RELATED WORK

Fuzzing Technique. Fuzzing is an automatic testing technique that covers numerous boundary cases using invalid data as input to ensure the absence of exploitable vulnerabilities [18]. Some popular AFL [34] family tools [2–4, 14, 17, 19, 20, 27, 29, 31, 35] apply various strategies to boost fuzzing process, including symbolic execution, scheduling algorithm and so on. For example, EnFuzz [4] integrates multiple fuzzing strategies to obtain better performance than that of any constituent fuzzer alone. There are also some tools focus on fuzzing in other domains, for example, QuanFuzz [30] is a search-based test input generator for the quantum programs.

Differential Testing. Differential testing [23] has been successful in uncovering differences between independent implementations with similar intended functionality. For example, Chen et. al perform differential testing of JVMs using MCMC sampling for input generation [5]. DLFuzz [13] was presented as the state-of-the-art differential fuzz testing framework for deep learning systems. It extended differential testing framework with the comparisons of multiple similar inputs, and does not need multiple platforms.

Smart Contract Validation. Smart contracts have been shown to be exposed to severe vulnerabilities [1, 15], and many efforts[16, 21, 26] have been devoted to ensure its correctness. For example, Luu et. al [21] designed Oyente, which builds the control-flow graph from the bytecode and then performs symbolic execution and checks whether there exist any vulnerable patterns. Zeus [16] is a sound analyzer that translates smart contracts to the LLVM framework and uses XACML as a language to write properties.

Main Difference. Different from the above work, EVMFuzzer mainly focuses on discovering the vulnerabilities in EVM. It takes the lead in paying attention to EVM security while others mainly concerned about smart contracts. Particularly, EVMFuzzer combines the basic ideas of fuzzing and takes advantage of EVMs’ multi-implementation to quickly find output discrepancies and reduce manual checking. Within EVMFuzzer, we also define the domain specific EVM test indicators to guide the differential fuzzing process with different contract mutation and selection strategies.

3 EVMFUZZER DESIGN

The overall workflow of EVMFuzzer is shown in Fig. 1, which consists of two major components, seed contract generation module and unified EVM execution module. EVMFuzzer is aim to apply differential fuzz testing on EVMs. It will continuously provide mutated smart contract to EVM platforms including target EVM and benchmark EVMs. These EVMs are then monitored for catching different execution output of target EVM and benchmark EVMs.

We design 8 mutators according to the logic features of the smart contract on three different granularity, including the word-level, character-level and statement-level. We maintain a priority queue based on the feedback metric difference. For a seed contract, we update the weight of corresponding mutator after the execution comparison. If the metric difference increases, the mutator ID is pushed into the queue in an descending order of the weight; otherwise, the queue will not update. Except for the weight update, we design five mutator combined strategies to further increase the randomness and diversity of mutation in each iteration, as detailed in [12].

Seed Prioritization and Selection. All the qualified seed contracts are stored in the seed contract pool. Based on the priority of the seed, we decide which seed to be mutated in a new iteration. In general, the contract that makes the metric difference among EVMs larger should be the candidates 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 can be selected.

EVMFuzzer is available at https://github.com/EVMFuzzer/EVMFuzzer

Figure 1: Overall workflow of EVMFuzzer, which mainly includes the seed generation module for guided contract generation and the unified execution module for information collection and cross-checking.

3.1 Seed Contract Generation

The seed contract generation module can be viewed as a test case generator. From Fig. 1, we can see that the seed contracts are stored in the seed pool. EVMFuzzer will rank the candidate contracts according to dynamic priority, and the contract in the first place will be selected for the next iteration. After choosing the contract for mutation, EVMFuzzer uses 8 predefined mutators and the combined strategy to guide the random generation. The goal is to generate contracts that can increase the degree of metric difference and trigger different execution output of target EVM and benchmark EVMs.

Seed Mutation. We design 8 mutators according to the logic features of the smart contract on three different granularity, including the word-level, character-level and statement-level. We maintain a priority queue based on the feedback metric difference. For a seed contract, we update the weight of corresponding mutator after the execution comparison. If the metric difference increases, the mutator ID is pushed into the queue in a descending order of the weight; otherwise, the queue will not update. Except for the weight update, we design five mutator combined strategies to further increase the randomness and diversity of mutation in each iteration, as detailed in [12].
3.2 Unified EVM Execution

EVM execution module provides a unified runtime environment for various EVMs. After receiving the contract file from the seed contract 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 EVMFuzzer 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, EVMFuzzer decides whether to put the seed contract into the seed pool where high-quality seeds preserved. Besides, when the execution output is inconsistent, this module will also record the potential exception for manual root cause analysis.

4 USING EVMFUZZER

4.1 Tool Implementation

We have implemented EVMFuzzer, as shown in Fig. 2. Specifically, EVMFuzzer provides a command line UI to interact with users as in Fig. 3. Currently, EVMFuzzer needs the target EVM’s source code or executable file and corresponding APIs. In the backend, EVMFuzzer integrates four widely used EVMs as the benchmark EVMs. Those four benchmark EVM are ethereumjs-vm v2.4.0 [10], py-evm v0.2.0-alpha.31 [9], aleth v1.5.0-alpha.6 [6] and geth v1.8.13 [8]. EVMFuzzer uses the solc 0.4.24 compiler [28] to generate seed contract’s EVM bytecode. The difference calculator is responsible for generating runtime difference among EVMs, which are used by Seeds Selector to determine whether to keep the seed contract or not and to update seed contract’s priority.

4.2 Running Example

Fig. 3 shows a screenshot of EVMFuzzer. We can see that EVMFuzzer first shows the version information and then lists the fuzz steps. It also displays the help information to let users know what to input. Since EVMFuzzer is an automated testing tool, users just need to put target EVM in the specified directory (“myEVM” folder), enter the API, and set the fuzz times, and then the EVMFuzzer will start. Here we construct a reinforced js-EVM which is able to stop dangerous transaction for the test.

5 PRELIMINARY EVALUATION

We use those four benchmark EVMs described in Section 4, for cross-validation, this section shows some preliminary evaluation results. Our initial seed contracts are the 36,295 real-world contracts which were crawled from the Etherscan [11]. All experiments were performed atop a machine with 8 cores (Intel i7-7700HQ @3.6GHz), 16GB of memory, and Ubuntu 16.04.4 as the host operating system.

Inconsistency among EVMs.

After the experiment, based on the two internal test indicators: \(d_{asUsed}\) and opcode sequence, we found large number of EVM discrepancies. The 33,424 contracts executed normally with the same opcode sequence are used for \(d_{asUsed}\) comparison, which excludes the inconsistency of gas consumption caused by different execution opcode sequences. Table 2 shows the number of contracts with \(d_{asUsed}\) inconsistencies. We can see that almost every platform has over 50% average inconsistency rate of \(d_{asUsed}\) with others, aleth even produces a different gas consumption over 90% of contracts.

We implemented this test EVM according to the idea of EVM*[22]
Table 1: Description of 5 high-risk vulnerabilities detected by EVMFuzzer.

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>platform</th>
<th>version</th>
<th>language</th>
<th>description</th>
<th>created date</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-18920</td>
<td>Py-EVM</td>
<td>v0.2.0-alpha.33</td>
<td>python</td>
<td>Py-EVM v0.2.0-alpha.33 allows attackers to make a vm.execute_bytecode call that triggers illegal values shown in stack.</td>
<td>20181103</td>
</tr>
<tr>
<td>CVE-2018-19183</td>
<td>js-evm</td>
<td>v2.4.0</td>
<td>JavaScript</td>
<td>ethereumjs-vm 2.4.0 allows attackers to cause a denial of service (vm.runCode failure and REVERT) via a &quot;code: Buffer.from(my_code, 'hex')&quot; attribute.</td>
<td>20181111</td>
</tr>
<tr>
<td>CVE-2018-19184</td>
<td>geth</td>
<td>v1.8.17</td>
<td>golang</td>
<td>cmd/evm/runner.go in Go Ethereum (aka geth) 1.8.17 allows attackers to cause a denial of service (SEGV) via crafted bytecode.</td>
<td>20181111</td>
</tr>
<tr>
<td>CVE-2018-19330</td>
<td>aleth</td>
<td>v1.5.0-alpha.6</td>
<td>cpp</td>
<td>** RESERVED ** Details would be public after the vulnerability has been repaired to avoid potential attack.</td>
<td>20181117</td>
</tr>
<tr>
<td>CVE-2019-7710</td>
<td>aleth</td>
<td>v1.5.0-alpha.7</td>
<td>cpp</td>
<td>** RESERVED ** Details would be public after the vulnerability has been repaired to avoid potential attack.</td>
<td>20190210</td>
</tr>
</tbody>
</table>

Table 2: gasUsed inconsistency.

<table>
<thead>
<tr>
<th></th>
<th>js-evm</th>
<th>Py-EVM</th>
<th>aleth</th>
<th>geth</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>js-evm</td>
<td>0</td>
<td>18115</td>
<td>31166</td>
<td>17486</td>
<td>66767</td>
</tr>
<tr>
<td>Py-EVM</td>
<td>18115</td>
<td>0</td>
<td>31176</td>
<td>1358</td>
<td>50649</td>
</tr>
<tr>
<td>aleth</td>
<td>31166</td>
<td>31176</td>
<td>0</td>
<td>31163</td>
<td>93505</td>
</tr>
<tr>
<td>geth</td>
<td>17486</td>
<td>1358</td>
<td>31163</td>
<td>0</td>
<td>50007</td>
</tr>
</tbody>
</table>

In total, 1,275 seed contracts were successfully executed on four EVM platforms and return the same output, but the sequence lengths were different. From Table 3, we can see that the sequence length of geth and aleth on these 1,275 contracts were always the same so we take it as the baseline. And after calculation, the length of the opcode sequence of js-evm is small and always below the baseline; but the length of the execution sequence of Py-EVM is large and always above the baseline.

Table 3: opcode inconsistency.

<table>
<thead>
<tr>
<th></th>
<th>js-evm</th>
<th>Py-EVM</th>
<th>aleth</th>
<th>geth</th>
</tr>
</thead>
<tbody>
<tr>
<td>js-evm</td>
<td>0</td>
<td>1275</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Py-EVM</td>
<td>1275</td>
<td>0</td>
<td>1240</td>
<td>1240</td>
</tr>
<tr>
<td>aleth</td>
<td>52</td>
<td>1240</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>geth</td>
<td>52</td>
<td>1240</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 outputs.

Vulnerabilities detected by EVMFuzzer. After discovering thousands of output inconsistencies, we conducted the manual analysis and tried to explore the root causes. We ensured its reproducibility and then carefully reviewed the source code of EVMs. Finally, we found defects in the EVM platforms, of which, 5 previously unknown 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, shown in Table 1.

We choose one of the CVEs for detailed elaboration. CVE-2018-19184 [25] is an execution segmentation violation that occurred on EVM of Go Ethereum (geth) [8]. The code associated with this vulnerability was in the cmd/evm folder, where the exception handling mechanism of EVM before geth v1.8.14 did not cover enough corner cases. Although the problematic code snippet is not the API that directly exposed to the end users, this problem can be exploited by malicious attackers to cause the denial of service.

Figure 5: Segmentation violation error on geth-evm v1.8.13.

6 CONCLUSION

In this paper, we propose EVMFuzzer, the first differential fuzz testing tool, to efficiently detect vulnerabilities of EVM implementations. EVMFuzzer introduces the definition of EVM fuzz testing metrics, gasUsed and opcode sequence, which measure the internal difference in execution information between EVMs. Besides, EVMFuzzer designs 8 mutators for smart contracts, so that it can generate plenty of seed contracts without syntax error in a short time. Under the guided seed generation and selection algorithm, EVMFuzzer shows strong defects mining capabilities.

We evaluated EVMFuzzer based on four widely used EVM implementations and conducted numerous mutation on 36,295 real-world smart contracts. Among the generated 253,153 smart contracts, more than half successfully showed the differential performance, including 1,596 variant contracts triggered inconsistent output results among the four EVM platforms. With manual root cause analysis, 5 vulnerabilities have been assigned with unique CVE IDs. Our future work mainly includes developing more general seeds mutators and conducting more extensive evaluations on more EVMs.
REFERENCES


