A Fuzz Testing Service for Assuring Smart Contracts

Mei, Xiupei; Ashraf, Imran; Jiang, Bo; Chan, W. K.

Published in:
2019 IEEE 19th International Conference on Software Quality, Reliability, and Security Companion, QRS-C

Published: 01/07/2019

Document Version:
Final Published version, also known as Publisher’s PDF, Publisher’s Final version or Version of Record

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1109/QRS-C.2019.00116

Publication details:

Citing this paper
Please note that where the full-text provided on CityU Scholars is the Post-print version (also known as Accepted Author Manuscript, Peer-reviewed or Author Final version), it may differ from the Final Published version. When citing, ensure that you check and use the publisher's definitive version for pagination and other details.

General rights
Copyright for the publications made accessible via the CityU Scholars portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights. Users may not further distribute the material or use it for any profit-making activity or commercial gain.

Publisher permission
Permission for previously published items are in accordance with publisher's copyright policies sourced from the SHERPA RoMEO database. Links to full text versions (either Published or Post-print) are only available if corresponding publishers allow open access.

Take down policy
Contact lbscholars@cityu.edu.hk if you believe that this document breaches copyright and provide us with details. We will remove access to the work immediately and investigate your claim.

Download date: 21/02/2022
A Fuzz Testing Service for Assuring Smart Contracts†

Xiupei Mei, Imran Ashraf
Department of Computer Science
City University of Hong Kong
Hong Kong
xpmei2-c@my.cityu.edu.hk
iashraf3-c@my.cityu.edu.hk

Bo Jiang
School of Computer Science and Engineering
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

W.K. Chan‡
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

Abstract—Smart contracts are code to keep agreements in the form of data or smart contracts residing in a blockchain system. Making data corruptions to these data records may lead to insert unwanted agreements, corrupt an existing one, or remove the latter ones mistakenly. This paper reports the progress of our work in developing a fuzz testing service with client-side support. It will present the overall approach of the testing services followed by a discussion on the road ahead.

Keywords—blockchain, fuzz testing, system architecture, security vulnerability, smart contract, Ethereum, smart contract

I. INTRODUCTION

Program testing is a cost-effective method to find out errors in the program code. It is widely practiced in the industry. Since Ethereum smart contracts [3] are in essence programs that can only be run deterministically and the size of the program execution is limited by the amount of gas that a transaction can carry [10], programs of this kind are considered easier to be implemented and simpler in logics. On the other hands, also owing to the limited amount of gas available to each transaction, the checking procedure enforced on some function call cannot be as comprehensive as the code counterparts (such as a Java program) executed in environments without similar gas restrictions. Thus, although these programs are considered simpler, yet developers may require using creative approaches or workarounds to codify the functionality intended to be expressed. These programs are thus intuitively more vulnerable. Although failing a function call of a smart contract will automatically cause the program state to revert to the state before calling the function, yet in some cases, a data corruption may occur without triggering a system failure. Finding such errors associating with critical data like agreements among smart contracts is necessary to improve the quality of these codes.

This work discusses our progress of developing a fuzz testing service. In summary, we have developed a state-of-the-art fuzz testing framework to test smart contract codes. Two particular innovations are in the testing of a set of smart contracts as a whole and the design of precise test oracles for several kinds of security vulnerabilities. The source code of this fuzzing component can be found online [2].

The above fuzzing uses a black-box approach to generate test cases. We are whitening the test case generation procedure by considering all sorts of artifacts, datasets and infrastructure. Owing to the update of Ethereum Solidity language, some classes of security vulnerabilities (such as reentrancy) have been depreciated. We are in the progress of updating the test oracle formulation.

The rest of this paper is organized as follows. Section II reviews the architecture of our testing service and its client support, and discusses the challenges. Section III summaries the progress presented in this paper.

II. A TESTING SERVICE

A. System Overview and Usage Scenario

The overall architecture of our testing service (Fuse) is summarized in Figure 1. There are two major components: a testing service and a client-side simulator. The former is responsible to test the given smart contracts and the latter is responsible to aid developers to debug and visualize the security vulnerability detected by the testing service component. The following scenario clarifies the purposes of these two components.

Scenario: Mary is a smart contract developer. She developed a piece of smart contract and has been tested using her own machine and development environment such as the unit testing framework in Remix or use some other tools with the Ganache blockchain simulation server. She then submits, after encryption, the compiled code of her smart contracts to our testing service via a web interface. Upon receiving the submission, the testing service decrypts the code and deploys the decrypted code on its blockchain in a sandbox. Then, a fuzzing generates test cases to fuzz test the code and monitors the program state changes and the smart contract’s interaction with other smart contracts. If there is any error with respect to the built-in test oracle of the fuzzing is found, the testing service prepares a test report by summarizing the test case sequence and organize the results.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on June 01,2020 at 06:15:14 UTC from IEEE Xplore. Restrictions apply.
B. Progress

We [1] have made progress for the following three modules shown in Figure 1: Fuzz Test Generation, Execution Profiling, and Vulnerability Detection. As discussed in Section I, we have formulated test oracles and are designing to incorporate the use of white-box information into the fuzz test case generation procedure. We have also checked how test oracles can be formulated and removed those depreciated ones. We are particularly interested in data corruption in relation to agreements, where we will set a priority to explore it further. Our idea is different from Wang et al. [11] in that they look for mismatch between a program variable and the digit token of a smart contract, which is a specialized form of data corruption. We aim to study a more general context in this project. We have formulated some test oracle conditions and realized the checking procedure as an offline dynamic analysis. We have proposed a novel form of fuzz testing with constant seeding as the basic idea to generate test transactions in which it fuzz-tests the whole set of deployed smart contracts, where these needed constants are extracted from this whole set. Our experiment results showed that the true positive rates were high. Compared to existing static analysis tools, the scope of our fuzz testing component can be restrictive, and yet in a recent study [8], researchers found that static analysers may only detect errors concentrating on small set of smart contracts. Table 1 shows the preliminary results of our fuzzer component [1], and its source code can be found at [2].

Table 1 Summary of Vulnerabilities Detected [1]

<table>
<thead>
<tr>
<th>Vulnerability Class</th>
<th># of Detected Vulnerabilities</th>
<th>True positive rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gasless Send</td>
<td>138</td>
<td>1.000</td>
</tr>
<tr>
<td>Exception Disorder</td>
<td>36</td>
<td>1.000</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>14</td>
<td>1.000</td>
</tr>
<tr>
<td>Timestamp Dependency</td>
<td>152</td>
<td>0.960</td>
</tr>
<tr>
<td>Block Number Dependency</td>
<td>82</td>
<td>0.965</td>
</tr>
<tr>
<td>Freezing Ether</td>
<td>30</td>
<td>1.000</td>
</tr>
<tr>
<td>Dangerous Delegatecall</td>
<td>7</td>
<td>1.000</td>
</tr>
</tbody>
</table>

C. Discussion

We still need to address a number of technical issues. Our fuzz testing prototype [2], even though precise in practice, are still restrictive. For instance, compared to Oynete [4][5], ContractFuzzer detected around 50% fewer true positive cases in the experiment [1]. This project will address the above issue by expanding the definitions of test oracle and incorporating a new testing technique in generating test data. SmartSCopy [9] employs a symbolic execution approach to generate partner smart contracts to help explore the vulnerability of a victim contract; where our preliminary work [2] is limited to these existing contracts. Both works are incomplete and can be very effective in penetrating through an encrypting function. A continuous testing or checking infrastructure, where the service we are building is an example, is desirable to aid developers during the lifespan of the smart contracts. Another issue is that the offline analysis could be inefficient. We plan to change a part of it into an online analysis. We have not stated designing the simulation component or the test report component.

III. Conclusion

This paper has presented the testing service architecture and discussed the progress and a plan for actions on Fuse.

REFERENCES