On the Need of Understanding the Failures of Smart Contracts

Dabao Wang*, Kui Liu†, Li Li*
*Monash University, Australia
†Nanjing University of Aeronautics and Astronautics, China

Abstract—Smart contracts, written as a piece of code, are designed to be run on blockchains as self-executing contracts with the terms of agreements between sellers and buyers. Like document-based contracts, it is vital to ensure the reliability of smart contracts, which requires automated analyzers. Unfortunately, when the execution of smart contracts fails, the transaction will not be recorded and hence cannot provide hints for analysts to improve their automated analyzers. To mitigate this, we present EexcuWatch to watch the execution of smart contracts and report the execution details, even when the execution of the contracts fails. For failed cases, EexcuWatch also attempts to infer the failure locations, aiming to provide hints for developers to better understand the failures. With a naive fuzzing approach, we experimentally show that EexcuWatch is useful and effective in achieving the aforementioned purposes.

1 Introduction

Blockchain, originally known as block chain, is an ingenious invention to deploy undeniable systems for recording data changes among different parties in a verifiable and permanent way. The data changes are essentially grouped into blocks that are further linked using cryptography (i.e., the cryptographic hash of a given block is stored by its immediate subsequent block). One thing that makes blockchain so promising to the practitioners, who have investigated millions of dollars in building the blockchain infrastructure, could be the innovation of smart contracts. Indeed, smart contracts provide means for the participants to execute a contract (such as exchanging money and shares) in a transparent, conflict-free manner. In this work, we limit ourselves to the Ethereum blockchain platform, which is the second most popular blockchain platform. The reason why we choose Ethereum instead of other blockchain platforms is that Ethereum weighs smart contracts as its strategic opportunity and positions itself as the internet of the future. The smart contracts running on the Ethereum platform are usually written via the so-called Solidity programming language, which is a super typed Javascript-like language with the inclusion of important object-oriented features such as inheritance.

At the end of 2018, there are already over a million smart contracts deployed on Ethereum, counting for a total of over 100 million Ether, the fundamental token of operation in Ethereum, or over 1.5 billion US dollars (i.e., each Ether is worth about over 150 US dollars at the time of writing). Unfortunately, where there is money, there are attackers following. Indeed, hackers have launched the infamous DAO attack[1] and have stolen at least 60 million US dollars from the Ethereum blockchain platform. More recently, the team behind the Parity Ethereum software client reveals that a critical code flaw (also known as the Parity Freeze[2]) has led to the freezing of around 160 million US dollar worth of Ether.

The fact that a security problem would lead to millions of dollar losses shows that it is essential to test smart contracts properly before releasing them. Indeed, state-of-the-art testing approaches have been proposed to mitigate potential security issues of smart contracts. For example, Jiang et al. [2] propose a prototype tool called ContractFuzzer, which applies fuzz testing to detect vulnerabilities in smart contracts. Unfortunately, state-of-the-art approaches ignore the failed test cases that could provide useful information for generating promising test cases. Indeed, various constraints can hinder the deployment or execution of smart contracts. If those constraints are not met, naïve fuzzing techniques could not bypass those constraints and subsequently may result in wasted fuzzing efforts. Therefore, there is a need to characterize the failures of smart contracts to achieve effective fuzzing.

Existing smart contract IDEs such as Remix provide debugging features that allow developers to execute the contracts step by step, so as to comprehend the contract failures, if any. However, such debugging processes are known as time-consuming, and most importantly, cannot be automated, which is nonetheless essential to achieve effective fuzzing. Indeed, it is non-trivial to automatically locate the failures of smart contracts under testing. The Ethereum virtual machine does not provide possible means to record the execution status of smart contracts, especially when a given smart contract is failed. Indeed, developers usually use the *event* system to log the execution of smart contracts. When the execution of a smart contract fails, all the execution status (even the already triggered events) will be rollbacked. As a result, when integrating execution feedback to improve fuzz testing approaches, there is a strong

*Manuscript received ***; revised ***.*

1. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

2. https://www.coindesk.com/parity-team-publishes-postmortem-160-million-ether-freeze

need to comprehend the failures caused by the consumed test cases.

In this work, we present a prototype tool called *ExecuWatch*[3], which leverages a code instrumentation approach to record the execution details of smart contracts, including the results of unsuccessfully executed contracts. We further leverage *ExecuWatch* to locate failures in smart contracts. When the execution of a given smart contract fails, *ExecuWatch* goes through the execution details to automatically locate the position where the failure happens, aiming at helping practitioners better understanding the reasons behind the failure so as to invent promising strategies to overcome such failures. Experimental results show that our approach is effective in logging the execution status of smart contracts and locating the failures of failed smart contracts. We also experimentally demonstrate that our approach helps invent effective fuzz approaches for testing smart contracts. Fuzz testing has been recurrently leveraged to automatically test software for identifying unexpected behaviors, crashes, and potential security issues [3], [4].

2 Motivation

We now motivate the importance of this work through a concrete example. Listing 1 illustrates a simplified example of a real-world smart contract written in Solidity. This example defines a contract named *Demo* (cf. line 1), which declares a field *inLedger* (cf. line 2), one event named *result* (cf. line 8), one constructor method (cf. lines 11), and three public methods defined via the *function* keyword (line 12, line13 and lines 14-30).

Although Solidity is similar to other programming languages, it has introduced several unique features that are worth highlighting. First, since there is no explicit *logging* mechanism introduced in Solidity, *events* are usually used to record the execution status of smart contracts. Second, observant readers may have already noticed that there is a public method declared without giving an explicit name (cf. line 12). This method is known as the fallback method, which will be triggered when the contract is called, but no methods match the calling signature. Third, modifiers of methods (such as public and payable) are defined at the end of arguments. Modifier *payable* indicates that the method is allowed to receive *Ether*s (the currency in the Ethereum ecosystem) from other contracts.

Traditional fuzzing involves generating random inputs, including unexpected or even invalid test cases, to explore the given software under testing. Unfortunately, due to various language features included in Solidity, it is is less effective to use traditional fuzz testing to test Etherum smart contracts. Take Listing 1 as an example, with a naïve fuzzing strategy, all the randomly generated test cases may fail to pass the method *bet()*. The main reason causing the failure of the fuzzing approach is related to the specific *require* related methods. The *require* statements define constraints that the value of the parameter expressions must be fulfilled.

Furthermore, smart contracts need to be first deployed on blockchains (or equivalent virtual machines that allow the emulation of contract deployment) before being executed. When addresses of other contracts are involved, in order to successfully run the smart contract, the referred addresses need to be valid as well (e.g., the corresponding smart contracts need to be deployed on the same blockchain).

3. *ExecuWatch* is publicly available at https://bitbucket.org/PanicWoo/execuwatch.

```
0  pragma solidity ^0.4.24;
1  contract Demo {
2   mapping(address => bool) inLedger;
3   mapping(address => uint256) balanceLedger;
4   uint256 max_bet_amount  = 1000;
5   uint256 min_bet_amount  = 100;
6   uint256 bonus_rate = 20;
7   event redeem_result(uint8 _guess, uint8 redeem_code);
8   event result(address player, uint256
        total_bonus,uint256 lost_amount);
9   event ReceiveFrom(uint,address);
10
11  constructor() public payable {...}
12  function () public payable{...}
13  function random(uint8 seed) public view returns
        (uint8) {...}
14  function bet(uint8 _guess, address player, uint256
        _amount, uint8 play_times) public payable {
15   uint256 total_bonus = 0;
16   uint256 lost_amount = 0;
17   uint256 bonus = bonus_rate * _amount;
18   require(inLedger[player], "Caller is not in the
        member list.");
19   for(uint8 i=0; i <= play_times; i++) {
20    emit result(player,total_bonus,lost_amount);
21    if (_guess == random(i)) {
22     player.transfer(bonus);
23     total_bonus = bonus + total_bonus;
24    }
25    else {
26     balanceLedger[player] -= _amount;
27     lost_amount = lost_amount + _amount;
28   }}
29   emit result(player,total_bonus,lost_amount);
30 }}
```

Listing 1: A simplified example of a smart contract written in Solidity.

The aforementioned challenges show that it is non-trivial to test smart contract with straightforward fuzzing. We argue that it would be more practical to take into account the execution results (specifically failures) as feedback to improving the fuzz testing approach. Unfortunately, in practice, it is difficult to achieve this purpose as there is no mean that can effectively collect the execution feedback of smart contracts under the current Ethereum execution environment. Indeed, when a smart contract fails to be executed, all the execution details, including the already fired events, will be rolled back. Therefore, to achieve effective fuzzing, it is necessary to effectively record the execution results of smart contracts, even if their executions are failed.

3 Approach

In this work, we design and implement a prototype tool called *ExecuWatch*, aiming at tracking the execution status of smart contracts (including failed ones), so as to help practitioners understand the failures of smart contracts, which are essential for effective fuzzing. Figure 1 illustrates the working process of *ExecuWatch* that consists of two main steps: code instrumentation and failure locating.

3.1 Code Instrumentation

The first step aims at injecting logging statements into the original smart contracts to record their execution statuses.

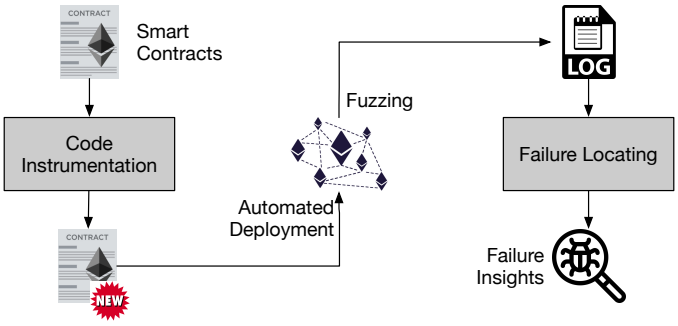

Fig. 1: The working process of *ExecuWatch*.

```
14 function bet(uint8 _guess, address player, uint256
       _amount, uint8 play_times) public payable {
15  // ...Hide previous lines
16    player.transfer(bonus);
17 +  if (targetLineNum == 9) {
18 +    logstring( 'Line' , "total_bonus = bonus +
        total_bonus;");
19 +    //logging global variables
20 +    loguint('_guess',_guess);
21 +    logaddress('player',player);
22 +    loguint('_amount',_amount);
23 +    loguint('play_times',play_times);
24 +    loguint('total_bonus',total_bonus);
25 +    loguint('lost_amount',lost_amount);
26 +    loguint('bonus',bonus);
27 +    return;
28 +  }
29    total_bonus = bonus + total_bonus;
30 }
```

Listing 2: The instrumented contract code for line 22 in Listing 1. The *log** functions are pre-defined (and also injected) by *ExecuWatch* to record the execution runtime.

By taking a smart contract as input, this step first leverages a lightweight static analysis approach to infer (1) what information to log and (2) where to inject logging statements. Based on the outputs of the static analysis approach, this step then applies a dedicated code re-writer to inject the previously inferred logging statements. As a result, this step will output a new smart contract that contains richer debugging information while being semantically equivalent to the original input contract. Listing 2 demonstrates such an example of instrumented code. All the '+' indicated lines are injected to record the execution status (inferred in this module to log) of line 22 in Listing 1.

3.2 Failure Locating

The second step is to locate the failure position. When the execution of a smart contract fails, the whole execution will be rollbacked to the initial state, and the emitted events will be emptied [1]. In other words, even with the injected logging statements, if the contracts' execution fails, we still cannot obtain the execution statuses of smart contracts. To this end, we invent a novel *divide-and-conquer* (D&C) strategy to bypass this challenge. The idea of D&C is to split the function to enable partial testing of the function, allowing the collection of partial execution statuses. For example, given a contract function that fails to be fully executed, we can divide the function into two parts with each part contains half of the statements. If the first half statements can be successfully executed, we will be able to harvest their execution records, and we are sure that the failure location is at the second half statements. In practice, this process will be automatically iterated until the failure point is located.

4 Evaluation

We now briefly detail the experiments that we carry out to assess *ExecuWatch* for locating and understanding failures in smart contracts.

In this work, we resort to real-world smart contracts deployed on Etherum blockchains to evaluate the performance of *ExecuWatch*. In particular, we collect smart contracts deployed from May to December in 2018 on Etherscan, one of the leading block explorers in the community, and randomly select 100 smart contracts to fulfill our evaluation dataset.

To investigate the locating failure capability of *ExecuWatch*, ideally, we should apply our approach to such smart contracts that have known failures with actual test inputs. Unfortunately, as smart contract research is at an earlier stage, our community (both practitioners and researchers) has not prepared such a ground truth to support our experiments. To this end, we conduct an experiment with a naïve fuzzing approach to automatically explore the execution of deployed smart contracts. For the sake of simplicity, we implement our fuzzing approach based on the strategy of ContactFuzzer, which is proposed by Jiang et al. [2] for detecting vulnerabilities in smart contracts. In this work, all the contracts are deployed and tested on a test blockchain set up via Geth[4], a golang implementation of Ethereum blockchain.

We apply the naive fuzzing approach to explore the randomly selected 100 smart contracts, containing 326 public functions. The fuzzing test for each function is lasted for 10 minutes. In total, more than 30,000 test cases are generated and tested, among which over half of them cannot pass the execution. By default, there will be no execution status generated for the failed cases, which could lead to difficulties in understanding those failures. With the help of *ExecuWatch*, all the executions, including the failed ones, have their execution details recorded. This evidence shows that *ExecuWatch* is indeed effective for "watching" the execution of smart contracts.

We then look at the capability of *ExecuWatch* for pinpointing the location of failures, aiming at providing hints for users and developers to quickly understand the reasons behind such failures. To this end, we randomly select 100 failed executions and manually go through all of them to check if the reported location is indeed relevant to the failure causes (e.g., the referred external contract, for which its address is hardcoded, is not deployed on the blockchains). Additionally, our in-depth investigation shows that 76% of them are correct results, illustrating that our approach is also useful in helping users understand the failures of smart contracts.

5 Implication and Discussion

When we manually check the located failure statements, we find that a significant number of them are caused by invalid test inputs. This is expected as we have only leveraged a

4. https://geth.ethereum.org/docs/

naïve fuzz testing approach to explore the contracts. Indeed, many failures are related to unsatisfied constraints (e.g., *require()* statements as shown in Listing 1), for which we believe a "smarter" fuzzing approach would bypass. To this end, based on the outputs of *ExecuWatch*, we go one step further to refine our approach by introducing a constraint-aware fuzzing approach. To this end, we first extract the related constraints from a smart contract before generating the inputs for testing. This simple improvement enables us to successfully pass more than 10% of smart contracts (i.e., between our simple constraint-aware fuzzing approach and the naïve fuzzing approaches), illustrating that *ExecuWatch* could be useful for guiding the development of fuzzing approaches. *ExecuWatch* enables an automated feedback mechanism when executing smart contracts, which could be essential towards developing effective fuzzing approaches. Indeed, fuzzing tools can leverage the feedback (i.e., execution runtime) yielded by *ExecuWatch* to dynamically update their test case generation strategy so as to generate more effective test cases, and subsequently lead to higher code coverages.

6 Conclusion

In this work, we presented to the community a prototype tool called *ExecuWatch*, which to the best of our knowledge, is the first approach proposed for recording the execution details of smart contracts, which by default, is impossible to harvest if the execution fails. Furthermore, for such contracts with failures, *ExecuWatch* adopts a failure locating module to pinpoint the location of the failures. The failure locations provide useful hints that could help users, developers, or analysts quickly understand the reasons behind. We further demonstrate that our approach is useful in supporting the development of advanced fuzzing approaches. As of our future work, based on *ExecuWatch*, we plan to implement such an advanced fuzzing approach to effectively testing smart contracts.

References

[1] Ethereum state transition function. https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function, Last Accessed: Oct. 2019.

[2] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018.

[3] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. *arXiv preprint arXiv:2004.08563*, 2020.

[4] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 116–126. IEEE, 2020.