Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 1565007, 8 pages
https://doi.org/10.1155/2022/1565007

Research Article

EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection

Xiaoyin Wang,[1,2,3] **Jiaze Sun**,[1,2,3] **Chunyang Hu**,[4] **Panpan Yu**,[1] **Bin Zhang**,[1] **and Donghai Hou**[1]

[1]*Xi'an University of Posts & Telecommunications, Xi'an Shaanxi 710121, China*
[2]*Shaanxi Provincial Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an Shaanxi 710121, China*
[3]*Xi'an Key Laboratory of Big Data and Intelligent Computing, Xi'an Shaanxi 710121, China*
[4]*Hubei University of Arts and Science, 441053 Xiangyang Hubei, China*

Correspondence should be addressed to Xiaoyin Wang; wangxiaoyinxy@126.com

Received 2 June 2022; Revised 17 July 2022; Accepted 30 July 2022; Published 26 August 2022

Academic Editor: Ruinian Li

With the development of Internet of Things technology, the use of Internet of Things is expanding, and its security risk will become an important factor restricting the development of Internet of Things technology. The application of blockchain technology in the security field of the Internet of Things can improve security problems, and the blockchain has immutable characteristics. Therefore, it is particularly important to ensure the security of blockchain smart contracts. However, the order of transaction in smart contracts is easy to be operated by miners, and there is a relative lack of tools to detect TOD (transaction-ordering dependent) vulnerabilities. The current smart contract vulnerability detection methods have the problems of low efficiency and low accuracy. Therefore, based on the study of TOD vulnerability principle, this paper creatively highlights a mutation fuzzy testing method EtherFuzz to specifically detect TOD vulnerability in smart contracts. Use the intelligent contract ABI (application binary interface) to generate test cases, test the byte code of the intelligent contract, use TOD to test oracle to detect TOD vulnerabilities, and then, mutate the tested data to generate new test cases. Finally, the behavior of smart contract operation is recorded, and the fuzzy test process is controlled until the vulnerability is detected. The experimental results show that when 987 token contracts are selected as Ethereum test objects, the false-positive rate, detection time overhead, and detection storage overhead of EtherFuzz are reduced by 74.4%, 30.1%, and 28.1%, respectively. Therefore, EtherFuzz has high speed, efficiency, and accuracy in detecting TOD vulnerabilities and has excellent application value.

1. Introduction

With the development of blockchain [1, 2] technology, blockchain is used in more and more scenarios. The emergence of Ethereum smart contract [3] has further expanded the scope of application. At present, smart contracts have been widely used in many fields, including digital payment [4], financial asset disposal, and cloud computing [5]. Due to the immutable nature of smart contracts, the security of smart contracts faces great challenges [6, 7]. The current definition of smart contract vulnerabilities has not yet formed a broad consensus. NCC group [8] concluded that there are 10 types of vulnerabilities which occur most frequently in smart contract, and these vulnerabilities contain reentrancy, TOD (transaction-ordering dependent), access control, arithmetic issues, unchecked return values for low level calls, denial of service, bad randomness, timestamp dependence, short address attack, and unknown unknowns. TOD is a kind of vulnerability occurring frequently in smart contracts, which has attracted wide attention.

The TOD is a common vulnerability in smart contracts. The transaction logic of Ethereum processing is that miners view the transactions they receive and select which transactions are included in the block according to who has paid a sufficiently high gas [9] price, and when the transactions are sent to the Ethereum network, they will be forwarded

to each node for processing. A block contains a collection of transactions, and the execution order of transactions belonging to the same block is uncertain (only miners can determine). Therefore, the status of the block is uncertain, resulting in TOD vulnerability. There is a TOD vulnerability in ERC20 token standard [10] for the approval and transmission of two-step transactions. Since most smart contracts comply with ERC20 standard, it is necessary to verify whether the contract contains TOD vulnerabilities through automated means. This vulnerability is obviously caused by the impossibility triangle of the blockchain: security can no longer be satisfied on the premise of decentralization and scalability. To put it more bluntly, it must take a certain amount of time for transactions to reach a consensus among many distributed nodes. In this long period of time, in order to improve the service throughput of the blockchain network, it is certain to receive multiple transactions in one block. However, there is no way to guarantee the specific transaction time sequence received by different nodes. Therefore, there is no way to guarantee the sequence of transactions executed in a block. Therefore, if several transactions based on the same contract occur in the same block, there is no way to determine the status of the contract.

Oyente [11] uses the dynamic symbol execution method [12] to detect TOD vulnerabilities. First, the Z3 solver [13] is used to search and mark the path containing the ether flow, return a group of traces and the corresponding ether flow of each trace, and check whether two different traces have different ether flows. If such trace air is included in the contract, Oyente reports it as a TOD contract. However, as the path depth increases, the constraints become more complex. Constraints are a challenge for constraint solvers to find solutions, which leads to high overhead and low execution efficiency.

Securify [14] uses the formal verification method [15] to detect TOD vulnerabilities, analyze the dependency graph of the contract, and extract accurate semantic information from the code. It then verifies TOD vulnerabilities by checking compliance and violation patterns. The compliance mode requires that the number of ethers sent by the call instruction is independent of the storage status and the balance of the smart contract account, which means that reordering transactions will not affect the amount sent by the call execution. The conflict mode checks whether the number of call instructions is determined by the value read from the store and whether the value can be updated. A TOD vulnerability exists if the conflicting mode is met. If the compliance mode is met, there is no TOD vulnerability. Since formal verification is a static detection method [16], a large number of false positives will be obtained. Therefore, the current TOD vulnerability detection methods have high false-positive rate and high overhead.

Fuzzy testing is a technology that can automatically and quickly generate test inputs and run them against the target program to find security vulnerabilities. Because of its simplicity and practicality, fuzzy testing has become one of the main methods of software testing [17]. Fuzzy testing of smart contract is an automatic testing technology, which uses random, unexpected, or invalid data as the input of the contract [18]. These input data are expected to lead to the detection of some unnecessary behaviors, such as crashes, some function failures, and permission errors. Sun et al. [19] used the fuzzy testing method based on the contract ABI and the user-defined method to detect TOD vulnerabilities, Wang et al. [20] used the smart contract control flow graph to design corresponding constraints and then used a feedback adjusted fuzzy testing method to detect gas-related vulnerabilities in smart contracts.

Mutation fuzzy test [21] is very simple and effective. It can improve path coverage without constraint solver and other overhead. At the same time, because fuzzing dynamically executes test data, the false-positive rate of vulnerability detection is low. Therefore, fuzzing can effectively solve the problems of high false-positive rate and high overhead [22].

In this paper, we propose EtherFuzz, a mutation fuzzing framework, to detect TOD vulnerability in smart contracts. EtherFuzz detects the TOD vulnerability according to the defined test oracle for TOD vulnerability through mutation amplification test data. At the same time, the test oracle for TOD vulnerability is defined as follows: the last two functions in the smart contracts are exchanged to generate the tested smart contracts with two different functions calling sequences.

If the smart contracts under test with different function call orders send different Ether, there exists TOD vulnerability. When the scale of the smart contract is large, the calling sequence of each function needs to be replaced when detecting the TOD vulnerability, and numerous different function calling sequences are generated, where its detection time is expensive and inefficient. Through studying the smart contract on Etherscan, it can be obtained that the first few functions in the smart contract mainly do some initialization operations, and most transfer operations exist in the last few functions. Subsequently, in order to reduce the detection time cost, only the last two functions in the smart contract are replaced in the test oracle defined. Mutation fuzzing smart contracts for TOD vulnerability detection proposed in this paper not only integrates the characteristics of low false-positive rate and low cost of fuzzing but also generates different function calling orders by changing the last two functions. In other aspects, this method can reduce the time cost of TOD vulnerability detection dramatically.

2. TOD Contract Mutation Fuzzer

In this section, we first gave an overview of our EtherFuzz tool. Then, we proceed to present the design of each core component of the tool in detail.

2.1. An Overview of EtherFuzz. An overview of EtherFuzz describing its workflow is illustrated in Figure 1. EtherFuzz is mainly made up of a preprocessing module and TOD vulnerability mutation fuzzing module. The preprocessing module mainly includes four steps: (1) compile the smart contract under test to generate the smart contract bytecode [23] and an ABI (application binary interface) [24]. (2) Generate test data based on the ABI of smart contracts [25]. (3) Instrument the original tested smart contract bytecode

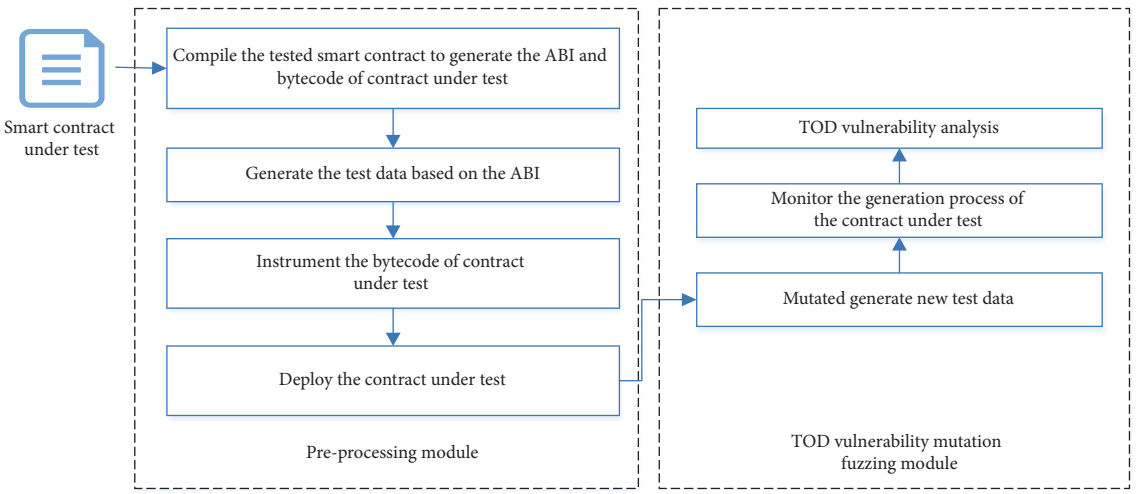

Figure 1: EtherFuzz overview.

and the tested smart contract bytecode after replacing the last two functions. (4) Deploy the smart contract under test. TOD vulnerability mutation fuzzing module contains three steps: (1) mutate the existing test data to produce new test data. (2) Monitor the fuzzing execution process of the contract under test to log the path coverage information, the value of the sent ether, and running result. (3) Analyze TOD vulnerability: if the running result is abnormal, the consequence of detecting TOD vulnerability will be outputted. If the running result is not abnormal, judging whether the test data cover the new branch or not, if the test data cover the new branch, the test data are saved and the mutation fuzzing is repeatedly executed.

2.2. Preprocessing Module. In this section, we will introduce the four steps of the preprocessing module in detail.

Step 1. Compile the smart contract to be tested, generate bytecode and ABI, and the bytecode is expressed in hexadecimal form, and it consists of three parts: deployment code, smart contract code, and smart contract encryption fingerprint.

Step 2. EtherFuzz generates test data by algorithm according to the user-defined smart contract, as shown in Algorithm 1. The input of the test data generation algorithm is interface. It is the ABI of the contract under test and then analyzes the interface to extract the data type of each functional parameter (line 1). If the data type of the smart contract function parameter is the address type, it is very important to generate a dedicated smart contract pool for the interface. Since the smart contract pool stores the addresses of all smart contracts called by functions in ABI, the addresses stored in the smart contract pool are randomly selected as test data (lines 2 to 5). If the function parameter type is a fixed size type, we can randomly select a set of values from the fixed size field of the parameter as test data (lines 6 to 8). If the function parameter type is nonfixed size, it will randomly generate a positive number as the length and

```
Input: Sinterface:the ABI of the contract under test
Output:test data
1:      dataType=analyze(Sinterface);
2:      if parameterType==addresType then
3:         contractPool=GeneratePool(Sinterface);
4:         testdata=ranselectAddress(contractPool);
5:      end
6:      if parameterType==fixedType then
7:         testdata=ranselectData(fixedSize);
8:      end
9:      if parameterType==nonfixedType then
10:        length=randomNumber();
11:           testdata=ranselectData(length);
12:     end
```

Algorithm 1: Test data generation algorithm.

randomly select a group of values from the randomly generated number field of the parameter as the test data (lines 9 to 12).

Step 3. Generate the tested smart contract with different function call sequences by exchanging the last two functions in the smart contract. After replacing the last two functions, we insert the bytecode of the smart contract in the original test and the smart contract in the test. We define that the trace_cov records an array of covering information of the current test data. And the virgin_cov records an array of global covering information, which represents the coverage information of all the test data that have been executed.

Step 4. The EVM (Ethereum virtual machine) [26] creates smart contract account and runs two deployment codes to test the smart contract after piling. In addition, it stores the smart contract code and the encrypted fingerprint of the smart contract on the blockchain. At the same time, it assigns their storage addresses to the code hash field in the smart contract account.

2.3. TOD Vulnerability Mutation Fuzzing Module. EtherFuzz uses a user-defined TOD vulnerability mutation fuzzy algorithm to detect TOD vulnerabilities in smart contracts. As shown in Algorithm 2, the inputs of TOD vulnerability mutation fuzzy algorithm are existing test data seeds and smart contracts deployed under test *P*. First, traverse and mutate the existing test data and the smart contracts deployed under the test to generate new test data (lines 3 to 4).

The mutation operations used are shown in Table 1, and they mainly include three types of mutation operations including the first type of mutation operation which is to flip bits or bytes, the second type of mutation operation which increases a randomly selected 1/2/4 bytes by a constant, and the third type of mutation operation which is to replace bytes according to different data types. Then, the deployed smart contract under test *P* executes new test data to log the path coverage information, the value of the sent ether, and running result (line 3).

Therefore, running result is obtained according to the defined TOD vulnerability test oracle. If the running result is TOD vulnerability and the test data triggering the TOD vulnerability is found, the detection result of "TOD vulnerability found" will be outputted (lines 4 to 6). Finally, it analyzes the coverage information of the recorded test data. If the test data cover a new branch, it will be added to the test data set, and the coverage information will be updated (lines 8 to 10). The process is repeated within the user-defined detection time.

```
Input: Seeds:the existing test data
P:deployed smart contract under test
Output:result:detect result
1     foreach seed ∈ Seeds do
2       foeach p ∈P do
3         newseed=Mutation(seed);
4         (trace_cov,ether,result)=Execute(P,newseed);
5       end
5     if testResult==hasTod then
6       result="TOD vulnerability found";
7     end
8     if hasNewCov(trace_cov,virgin_cov) then
9       Seeds=Seeds ∪ {newseed};
10        update(vrigin_cov,trace_cov);
11    end
12  end
```

Algorithm 2: TOD vulnerability mutation fuzzing algorithm.

3. Experiment and Result Analysis

In this section, we evaluate EtherFuzz through multiple experiments. The experiments are intended to confirm the following research questions (RQ).

RQ1: what is the efficiency of the TOD vulnerability detection method that only exchanges the last two functions?

RQ2: what is the false-positive rate of EtherFuzz in detecting the TOD vulnerability? Compared with existing vulnerability detection tools, is the false-positive rate of EtherFuzz reduced?

RQ3: what is the detection costs of EtherFuzz for detecting the TOD vulnerability? Is EtherFuzz's TOD vulnerability detection cost less than the existing detection methods?

Our test subjects include the 987 ERC20 token contracts which we have crawled from Etherscan. All experimental results reported below are obtained on an Ubuntu 18.04.1 LTS machine with Intel Core i7 and 16 GB of memory. EtherFuzz firstly performs static analysis on each contract to develop the private contract pool for each ABI interface and to extract the ABI functions. With the test data generation algorithm, EtherFuzz proceeds to generate input data.

3.1. Analyze the TOD Vulnerability Detection Method That Only Exchanges the Last Two Functions

3.1.1. Experiment I. We compared two methods of TOD vulnerability detection in the exchange order for functions. The first is the TOD vulnerability detection method used in EtherFuzz that only exchanges the last two functions, and the second is the TOD vulnerability detection method that exchanges all functions (simply change the TOD vulnerability detection method in this paper to reverse the order in which all functions are called, and we can get the second detection method). In this experiment, depending on the two TOD vulnerability detection methods, we detected the 987 ERC20 token contracts. It is assumed that we generate m groups of test data and n contracts under test. The first detection method generated two groups of smart contracts under test, so the time complexity of the TOD vulnerability detection algorithm is $O(m)$. And the second detection method generated $n!$ groups of contracts under test; therefore, the time complexity of the TOD vulnerability detection algorithm is $O(m * n!)$. The time complexity of the original detection algorithm is much lower than that of the second detection algorithm. Table 2 displays the correct detection number, false-positive numbers, and the total detection time. Meanwhile, TW represents the correct detection number, and FW represents the false-positive number.

As shown in Table 2, the accuracy of the first detection method and the second detection method is 12.4% and 12.9%, respectively, among which the false-positives of the two detection methods are 2.38% and 2.29%, respectively, and the average detection time of the two methods is 2.5 s and 3.3 s, respectively.

The results of this experiment show that the accuracy of the first method decreased by 3.8%, and the false-alarm rate increased by 3.9%; however, the average detection time of the first method decreased by 24.2%. Therefore, the experimental data shows that the efficiency of the former method was greater than of the second detection method; thus, we use the first method that exchanges the last two functions in EtherFuzz.

3.2. Analysis on False-Positive Rate of Smart Contract TOD Vulnerability Detection

3.2.1. Experiment II. We compared EtherFuzz with Oyente and Securify on the detection of the 987 ERC20 token

Table 1: Mutation operations.

Mutation operations	Detail
singleFlipBit, twoFlipBit, and fourFlipBit	Flip a randomly selected 1/2/4 consecutive bits
singleFlipByte, twoFlipByte, and fourFlipByte	Flip a randomly selected 1/2/4 consecutive bytes
singleIncrease, twoIncrease, and fourIncrease	Increase a random selected 1/2/4 bytes by a constant
singleSubstitute, twoSubstitute, and fourSubstitute	Replace randomly selected 1/2/4 bytes with special constants
overwriteWithDictionary	Replace a value at random with a constant from the smart contract
overwriteWithAddressDictionary	Replace the address and balance in the test data with the randomly generated address and balance

Table 2: TOD vulnerability detection results of two detection methods.

Method	TW	FW	Total detection time (min)
Exchange the last two functions	123	3	41
Exchange all the functions	128	3	54

Table 3: TOD vulnerability detection results of three detection tools.

TOD detection tool	TW	FW
Oyente	120	6
Securify	135	14
EtherFuzz	122	3

contracts, and the detection time is set to 150 minutes. Table 3 shows the correct detection number and the false-positive number.

Shown in Table 3 is Securify's smart contract TOD vulnerability detection for the most number of correct detection and the most false-positive. The number of EtherFuzz's smart contract TOD vulnerability detection false-positives is the lowest. Through calculation, the false-positive rate of the three tools is illustrated in Figure 2.

As shown in Figure 2, Oyente, Securify, and EtherFuzz have alarmed rates of 4.8%, 9.4%, and 2.4%. Securify has the highest false-positive rate, and EtherFuzz has a significantly lower false-positive rate.

This experiment shows that EtherFuzz can effectively reduce the false-positive rate of TOD vulnerability detection, and the false-positive rate decreases by 74.4%.

3.2.2. Experiment III. According to the detection results of 987 smart contracts under test by EtherFuzz, we select a contract under test with TOD vulnerability and a contract under test without TOD vulnerability. As the detection time is 240 s, the two contracts are detected in EtherFuzz, and the number of the covering branches of the two contracts at a different time is reported as shown in Figure 3.

As shown in Figure 3, the number of covering branches of the smart contract under test with TOD vulnerabilities gradually increases within 100 s-180 s, and it remained stable within 180 s-240 s. The number of covering branches of smart contracts without TOD vulnerabilities gradually increased within 100 s-220 s and stabilized within 220 s-240 s. Because of EtherFuzz's TOD vulnerability mutation fuzzing algorithm to mutate the test data, if TOD vulnerability is not detected, it will continuously generate test data covering the new branch. We note that EtherFuzz could detect this vulnerability in 180 s, so it will no longer mutate to generate additional test data from 180 s to 240 s. The branch coverage of the test data slightly increased.

Because TOD vulnerability mutation fuzzing algorithm generates new test data to continuously cover new branches through mutation and dynamically executes the smart contract under test, therefore, it is effective for EtherFuzz to reduce the false-positive rate.

The above two experiments show that EtherFuzz can effectively reduce the false-positive rate of TOD vulnerability detection, and the false-positive rate is reduced by 74.4%.

3.3. Cost Analysis of TOD Vulnerability Detection

3.3.1. Experiment IV. We, respectively, selected the smart contracts with correct detection of TOD vulnerability of each tool in Experiment II, in which 120 TOD smart contracts correctly detected by Oyente, 135 TOD vulnerability contracts correctly detected by Securify, and 122 TOD vulnerability contracts correctly detected by EtherFuzz. Then, we marked them as a smart contract under test A, B, and C and detect them separately in the corresponding detection tool. Then, we recorded the total detection time and total memory cost. The TOD vulnerability detection time and memory cost of the three detection tools are shown in Table 4.

As shown in Table 4, the detection cost of EtherFuzz is significantly reduced. The detection time cost and memory cost of EtherFuzz are reduced by 30.1% and 28.1%, respectively.

3.3.2. Experiment V. In this experiment, firstly, we analyze the 120 contracts that can be detected by the three tools, including 7 combinations which contains 6 contracts with

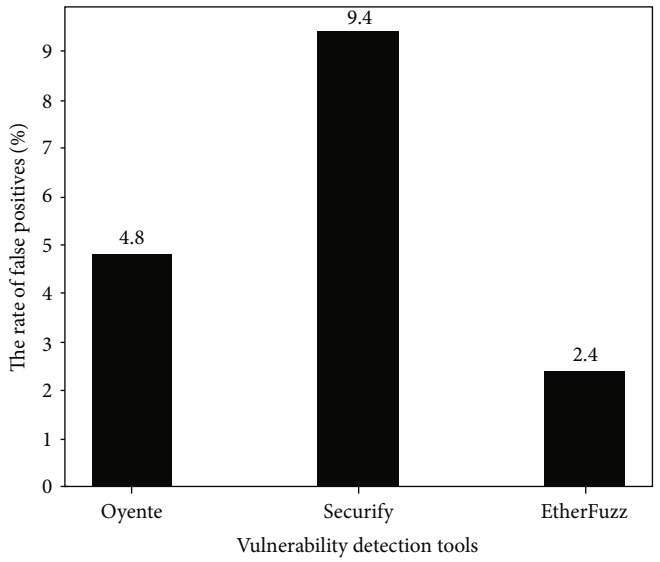

Figure 2: The false-positive rate of the three tools.

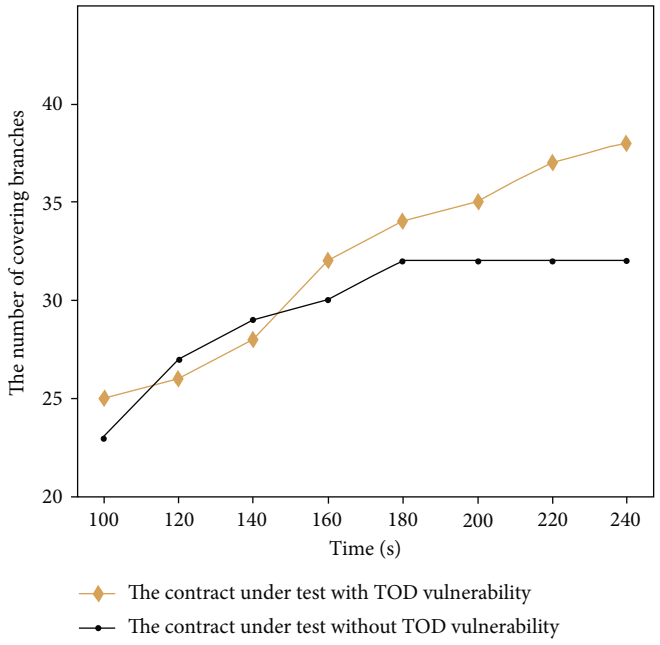

Figure 3: The number of the covering branches of two smart contracts under test.

Table 4: Analysis of detection cost of the three tools.

TOD vulnerability detection tool	Time cost (s)	Memory overhead (MB)
Oyente	732	1272
Securify	702	1215
EtherFuzz	512	915

2 transactions, 16 contracts with 3 transactions, 18 contracts with 4 transactions, 21 contracts with 5 transactions, 19 contracts with 6 transactions, 22 contracts with 7 transactions, and 18 contracts with 8 transactions. At the same time, they are marked as A2, A3, A4, A5, A6, A7, and A8, respectively (the transactions selected in smart contract under test do not include the function with "return," because the function with "return" does not change the state of the smart contract, so there is necessary to detect it). Then, we use three tools to detect 7 combinations, respectively, and record their detection time. The average detection time of 7 combinations is shown in Figure 4.

As shown in Figure 4, when the number of transactions in the contract is small (the number of transactions is less than 4), the detection time gap between the three tools is small, but with the increase of the number of contracts, the detection time gap among the three tools gradually

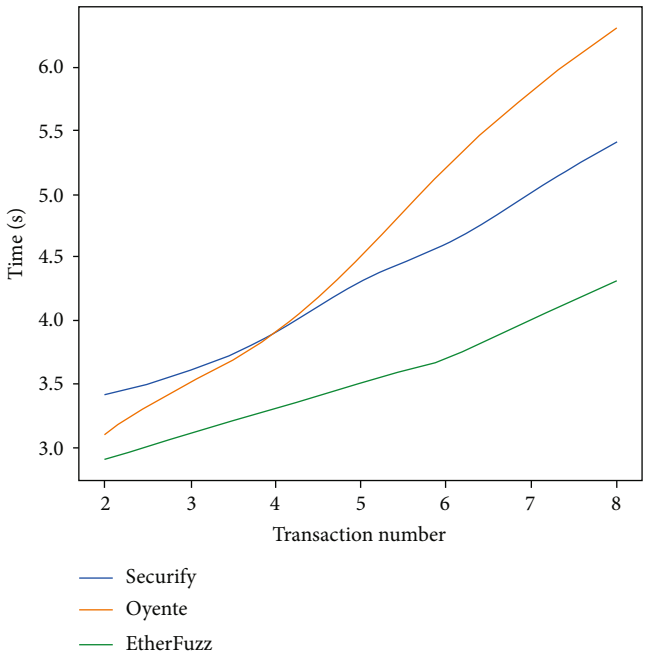

Figure 4: The detection time of the four contracts under test by three tools.

increases, and the detection time of EtherFuzz is significantly lower than that of the other two detection tools. At the same time, because EtherFuzz uses TOD vulnerability detection algorithm which only replaces the last two functions, it can be seen from the figure that with the increase of the number of contract transactions, the detection time of Oyente and Securify increases rapidly, while the detection time of EtherFuzz increases relatively smoothly.

To sum up, when compared with Oyente and Securify, the time cost and memory cost of EtherFuzz to test TOD vulnerabilities are significantly reduced. Among them, the average time cost is reduced by about 31.1%, and the average memory cost is reduced by about 29.2%. Moreover, the detection algorithm in EtherFuzz has significant advantages in terms of detection time for the larger smart contracts.

4. Conclusion

In this paper, we propose a variant fuzzy framework EtherFuzz to detect TOD vulnerabilities in smart contracts. In the smart contract, only the last two function call sequences are exchanged, and the mutation generates new test data, covering the new branch. The experimental results show that EtherFuzz can identify TOD vulnerabilities more accurately than other tools. Among them, the average time cost is reduced by about 31.1%, and the average memory cost is reduced by about 29.2%. Therefore, EtherFuzz can significantly reduce the false-positive rate and detection cost and can detect larger smart contracts more effectively. At the same time, it can also effectively defend and deal with security issues in the Internet of Things. Our future work will focus on tool optimization: (1) more seed variant designs. Designing more complex seed mutation functions can better detect vulnerabilities in smart contracts. (2) More efficient seed priority scheduling. The scale and storage structure of seed mutation still need to be optimized to further improve the efficiency. Optimize the detection algorithm in EtherFuzz to reduce the false-positive rate of TOD vulnerabilities, and try to use this tool to detect more related vulnerabilities in smart contracts.

Data Availability

The raw/processed data required to reproduce these findings cannot be shared as the data contains private data.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (Grant No. 61876138), the Key R&D Project of Shaanxi Province (2020GY-010), the Key Industrial Chain Core Technology Research Project of Xi'an (2022JH-RGZN-0028), and the Special Fund for Key Discipline Construction of General Institutions of Higher Learning from Shaanxi Province.

References

[1] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: past, present, and future trends," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–41, 2022.

[2] Z. Liu, N. C. Luong, W. Wang et al., "A survey on blockchain: a game theoretical perspective," *IEEE Access*, vol. 7, pp. 47615–47643, 2019.

[3] Z. Zheng and X. Chen, "An overview on smart contracts: challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[4] X. Ye, K. Sigalov, and M. König, "Integrating BIM-and cost-included information container with blockchain for construction automated payment using billing model and smart contracts," *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, vol. 37, pp. 1388–1395, 2020.

[5] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: applications, challenges, and future trends," *Peer-to-peer Networking and Applications*, vol. 14, no. 5, pp. 2901–2925, 2021.

[6] W. Xiong and L. Xiong, "Data trading certification based on consortium blockchain and smart contracts," *IEEE Access*, vol. 9, pp. 3482–3496, 2020.

[7] S. Rouhani and D. Ralph, "Security, performance, and applications of smart contracts: a systematic survey," *IEEE Access*, vol. 7, pp. 50759–50779, 2019.

[8] NCC Group, "Decentralized application security project top 10 of 2018," Jul. 2018. https://dasp.co/.

[9] S. M. Werner, P. J. Pritz, and D. Perez, "Step on the gas? a better approach for recommending the Ethereum gas price," in *Mathematical Research for Blockchain Economy*, pp. 161–177, Springer, Cham, 2020.

[10] R. Rahimian, S. Eskandari, and J. Clark, "Resolving the Multiple Withdrawal Attack on ERC20 Tokens," in *2019 IEEE European symposium on security and privacy workshops (EuroS&PW)*, pp. 320–329, Stockholm, Sweden, 2019.

[11] L. Loi and H. Aquinas, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, Vienna, Austria, 2018.

[12] Z. Tian, "Smart contract defect detection based on parallel symbolic execution," in *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, pp. 127–132, Nanjing, China, 2019.

[13] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "RA: hunting for re-entrancy attacks in Ethereum smart contracts via static analysis," in *2020 IEEE International Conference on Blockchain (Blockchain)*, pp. 327–336, Rhodes, Greece, 2020.

[14] T. Petar and V. Martin, "Securify: practical security analysis of smart contracts," in *In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82, Toronto, Canada, 2018.

[15] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *Australasian Computer Science Week Multiconference*, pp. 1–10, New York, 2021.

[16] F. Henglein, C. K. Larsen, and A. Murawska, "A formally verified static analysis framework for compositional contracts," in *International Conference on Financial Cryptography and Data Security*, pp. 599–619, Springer, Cham, 2020.

[17] S. Pani, H. V. Nallagonda, S. Prakash, R. Vigneswaran, R. K. Medicherla, and M. A. Rajan, "Smart contract fuzzing for enterprises: the language agnostic way," in *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, Bangalore, India, 2022IEEE.

[18] M. Böhme and B. Falk, "Fuzzing: on the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 713–724, New York, 2020.

[19] J. Sun, P. Yu, and B. Zhang, "Mutation fuzzy detection of TOD vulnerability in smart contract," in *The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, pp. 1687–1694, Springer, Cham, 2021.

[20] X. Wang, D. Hou, C. Tang, and S. Lv, "A fuzzy testing method for gas-related vulnerability detection in smart contracts," in *The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, pp. 407–418, Springer, Cham, 2022.

[21] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 557–560, New York, 2020.

[22] J. Sun, J. Deng, Y. Li, and N. Han, "A BCS-GDE multi-objective optimization algorithm for combined cooling, heating and power model with decision strategies," *Applied Thermal Engineering*, vol. 213, article 118685, 2022.

[23] J. Huang, S. Han, W. You et al., "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[24] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the Internet of Things," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1594–1605, 2019.

[25] G. Zheng, L. Gao, L. Huang et al., "Application binary interface (ABI)," in *Ethereum Smart Contract Development in Solidity*, pp. 139–158, Springer, Singapore, 2021.

[26] W. C. Yang and J. Peng, "Research on EVM-based smart contract runtime self-protection technology framework," in *Workshops of the International Conference on Advanced Information Networking and Applications*, pp. 617–627, Springer, Cham, 2020.