Journal of Quantum Computing JQC, vol.2, no.1, pp.11-24, 2020

Smart Contract Fuzzing Based on Taint Analysis and Genetic Algorithms

Zaoyu Wei[1, *], Jiaqi Wang[2], Xueqi Shen[1] and Qun Luo[1]

Abstract: Smart contract has greatly improved the services and capabilities of blockchain, but it has become the weakest link of blockchain security because of its code nature. Therefore, efficient vulnerability detection of smart contract is the key to ensure the security of blockchain system. Oriented to Ethereum smart contract, the study solves the problems of redundant input and low coverage in the smart contract fuzz. In this paper, a taint analysis method based on EVM is proposed to reduce the invalid input, a dangerous operation database is designed to identify the dangerous input, and genetic algorithm is used to optimize the code coverage of the input, which construct the fuzzing framework for smart contract together. Finally, by comparing Oyente and ContractFuzzer, the performance and efficiency of the framework are proved.

Keywords: Smart contract, fuzzing, taint analysis, genetic algorithms.

1 Introduction

Blockchain is a distributed technology that establishes trust relationship under the condition of deintermediation, which has the characteristics of high transparency, decentralization, tamper resistance, traceability, etc., [Nakamoto (2009)]. It is considered to be a strong support to reconstruct the system of credit, value and network. Although the concept of smart contracts is much earlier than blockchain technology, smart contracts exist only in theory for a long time with the lack of a credible execution environment, which is not applied to the actual industry. It is because of the blockchain's highly transparency, decentralization, and tamper resistance that blockchain can perfectly meet the running environment requirements and provide a reliable platform for the execution for smart contracts.

Ethereum [Chris (2019)] has creatively applied smart contracts to blockchain, and it has become the most widely used blockchain application scenario up to now, which represents the development of blockchain reaching a new stage.

Smart contract has greatly improved the services and capabilities of blockchain, but it has become the weakest link of blockchain security because of its code nature. Security

[1] School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, 100876, China.

[2] College of New Media, Beijing Institute of Graphic Communication, Beijing, 102600, China.

[*] Corresponding Author: Zaoyu Wei. Email: weizaoyu2017@bupt.edu.cn.

Received: 07 January 2020; Accepted: 09 March 2020.

incidents caused by smart contract vulnerabilities emerged in an almost endless stream. In June 2016, the reentrancy of The DAO led to a loss of 50 million US dollars and the hard fork of Ethereum; In July 2017, due to the vulnerability of Parity's multi-signature library, attackers could obtain others' wallets ownership by initializing the wallets, so that the assets of the legitimate user cannot be transferred, which eventually resulted in nearly 152 million US dollars being frozen; In April 2018, the BEC was suffered from the integer overflow, and its value almost was made instantaneously into zero.

In the past research, several smart contract vulnerability detection tools were proposed [Jiang, Li and Chan (2018); Luu, Chu and Olickel (2016)], but they have certain limitations. First, the accuracy of their detection strategies is not high enough, which results in an unsatisfied false alarm rate. Second, there are efficiency problems that the code coverage cannot attach balance with the running efficiency. Therefore, this paper is oriented to Ethereum smart contract, using taint analysis and genetic algorithm to optimize the input data of fuzzing, so as to improve the performance and efficiency of smart contract fuzzing.

2 Algorithm and module

2.1 Framework

The vulnerability detection method used for the Ethereum smart contract in this paper is fuzzing. However, when the original inputs are generated by the traditional fuzzing [Michael, Adam and Pedram (2007)], all the bytes of inputs need to be mutated blindly, which results in a large number of invalid test cases, reducing the efficiency of the detection. Therefore, the main work of this paper is to optimize and filter the generation of fuzzing inputs for the Ethereum smart contract, thereby improving efficiency and coverage.

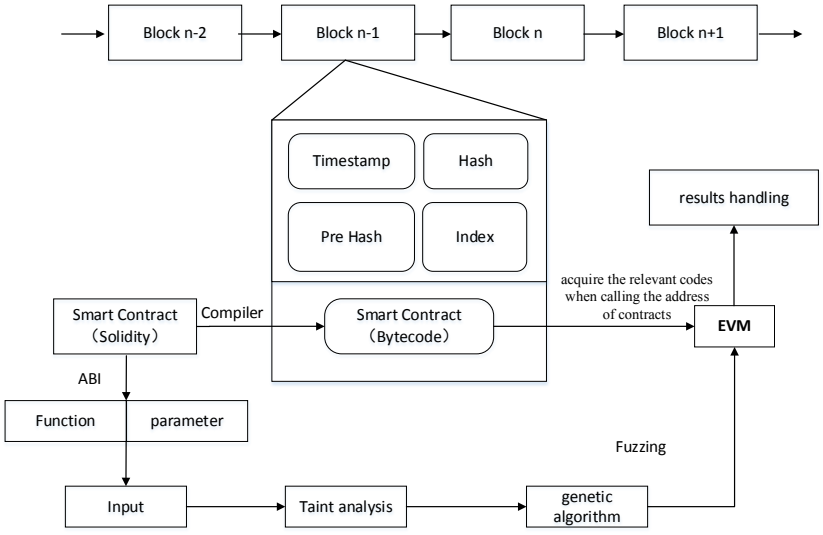

Figure 1: The framework of smart contract fuzzing based on taint analysis and genetic algorithms

The upper part of the framework is mainly the operating structure of the Ethereum smart contract. In Ethereum, the running and calling of smart contracts are completed through the EVM (Ethereum Virtual Machine). When a smart contract needs to be called, Ethereum will get the corresponding code from the block according to the contract address, and then load it into the EVM to run. Smart contracts can be developed when using the high-level language, Solidity, but the contract source code must be compiled with bytecode to run in the EVM. When smart contracts are deployed and interacted in the EVM, bytecodes are passed and presented as hexadecimal strings.

The rest parts of this framework are mainly related to the optimization processing of fuzzing inputs. The basic idea is to add two selecting sessions, taint analysis and genetic algorithm. First, in the taint analysis classification stage, the inputs is divided into three categories: coverage data, dangerous data, and harmless data. By ignoring the harmless data, the inputs that need to be mutated in the fuzzing process are reduced. Besides, according to the mutation of population selection in genetic algorithm, the coverage data is expanded in a certain scale. That is, the input is filtered and optimized. Finally, the system performance will be compared with the current widely used smart contract vulnerability detection tools and given a conclusion.

2.2 Taint analysis

It is the EVM that the taint analysis proposed in this section applies to, but there is no open literature to develop the Ethereum's taint analysis research. Therefore, we intend to monitor the taint propagation process when the smart contract is running, by instrumenting the EVM. Because the EVM follows the stack machine architecture, most of the instructions are to operate on the data in the stack. General taint analysis methods or tools cannot be applied to Ethereum smart contracts well, so the method of taint analysis for smart contracts in the EVM is one of the key points of research.

So, we first studied the feasibility of taint analysis for the EVM. According to the analysis of the source code of Ethereum [Grishchenko, Maffei and Schneidewind (2018)], there are neither no registers in the EVM nor no IO related instructions. The operations are only performed among memory, storage, and stack. The role of memory is mainly used to store the return value of the function. The storage is analogous to the computer hardware, mainly used to store global variables and contract status. And the stack is mainly used to store local variables. The stack machine architecture leads to a fact that most of the instructions are to obtain parameters through the stack, and then return results of the instructions are also directly stored in the stack. Obviously, our main focus is on taint tracking in the stack. The EVM architecture is completely deterministic, and smart contracts in EVM can only access blockchain state using dedicated instructions. That is, no other form of input or output is possible, which allows us to fully model the system's taint propagation process by tracking the taint data stream at the EVM instruction level.

Second, we need to determine the position of the instrumentation in EVM. The execution of smart contracts is in bytes, each byte represents an EVM instruction or an operation data, so the number of instructions in the EVM is only 256 at most (0×0~0×ff). Any of these instructions must contain the following attributes which we must pay attention to

when instrumenting: the instruction code; the number of elements pushed onto the stack after the instruction is executed; the gas that implements the instruction. Besides, we can see that EVM defines the instruction code in opcode.go and implements the processing of the instruction in the instructions.go, according to its code structure. Therefore, we mainly modified the EVM bytecode interpreter (instructions.go) to adapt to the taint analysis to monitor the running of smart contracts.

The specific taint analysis implementation process is shown in the figure below. Dynamic taint analysis has the same rules in the implementation process. Therefore, this paper draws on [Bekrar, Bekrar, Groz et al. (2012)] in the main process and makes certain adaptive adjustments to each process for EVM.

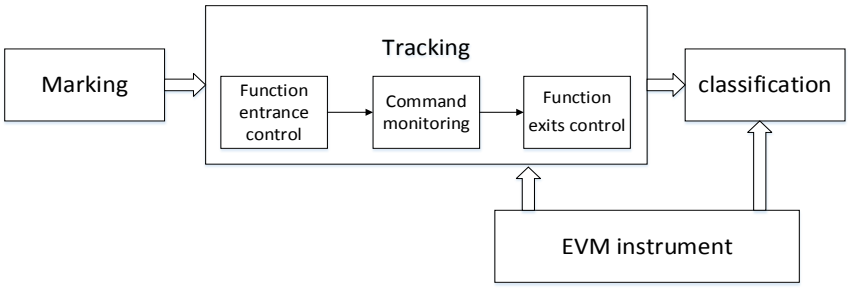

Figure 2: The process of taint analysis

This method has the following three processes:

(1) Marking. According to Section 3.1, the raw data of the fuzzing is generated according to the data type of parameters, so for each data type of each function under the smart contract, we will generate a tainted variable. Each byte is marked with a bit of taint, where "0" represents clean, and "1" represents taint, and the taint information is stored by the array taint_map.

(2) Tracking. When tracking, we control the overall execution environment through the function entrance and exit and analyze the taint propagation process of the code execution by instrumenting. And the data movement instruction (such as *jump*) or arithmetic instruction (such as *add*) will be monitored. Once the instruction is detected as a taint source, the taint diffusion occurs, and the instruction purpose (internal variable) is also marked as a taint variable.

(3) Classification. When the tainted variable is used as the conditional jump instruction parameter (only *jumpi*), the data corresponding to the tainted variable is classified into the overlay data; when the tainted variable is used as the parameter of the dangerous operation, the corresponding data is classified as dangerous data; the remaining data is classified as harmless data.

2.3 Dangerous operation library design

According to the previous section, the definition and identification of dangerous operations related to known vulnerabilities is also one of the key tasks of this paper. In this section, we analyze the dangerous operations involved in DASP's smart contract of Top10 vulnerability and then design a dangerous operation library [Kalra, Goel, Dhawan

(2018)]. When a taint variable is used as a parameter for the following operation, or when a dangerous operation occurs during execution, it is classified as a dangerous data.

Gasless Send. This vulnerability is mainly found in the send(). The smart contract uses the send() to send ether to the receiver in Ethereum, but when the send() function fails to execute, the fallback() function of the receiving contract will be called to roll back ether. Under conditions that the sending amount of Ethereum is non-zero and the default limit of gas consumed by the fallback() is 2300, when the receiving contract specifically sets the fallback() to consume much more than 2300 gas, the sending contract will trigger out-of-gas indicating what is abnormal. If this out-of-gas exception is not detected, gasless send will be caused.

We use send() as a dangerous operation needy to detect. In the EVM instruction, send() is implemented as a special type of call() and is needed to verify whether the input is 0 and the gas limit is 2300 or not, and whether the error code of out of gas is returned during execution.

The following codes are for dangerous operation detection of gasless send, and the detection of other vulnerabilities is similar:

Table 1: Detection code of gasless send

```
func (danger_op *GaslessSend) Exceptbool () bool{
        ifException:= false
        calls:= danger_op.taint_calls [0].nextcalls
        for call:= range calls{
                if danger_op.TriggerExceptionCall (call){
                        danger_op.taint_exception_calls =
append(danger_op.taint_exception_calls, call)
                        ifException = true
                }
        }
        return ifException;
}
func (danger_op * GaslessSend) TriggerExceptionCall(call *ContractCall) bool{
        return
IsAccountAddress(call.callee)&&call.throwException==true&&call.errOutGas==tru
e&&len(call.input)==0&&call.gas.Uint64()==2300
}
```

Reentrancy. A reentrancy vulnerability means that the called function may be executed multiple times before the operation of calling other functions. An operation that calls an external contract or sends ether to a specified address requires the contract to submit an external call. However, if the external call is hijacked by the attacker, it will force the contract to further execute other malicious code, such as callback of itself, which is

equivalent to the code re-entering the contract so as to obtain ether repeatedly. At this point, the status of the victim contract will change. Therefore, as long as one smart contract (represented as the victim contract) whose status of the contract is updated calls another smart contract (represented as the attacker contract), and the victim contract will perform operations on the update status, it can be judged that there is a dangerous operation of reentrancy.

Timestamp Dependency and Random Number Generation. Each block in the blockchain is identified with a specific timestamp, which is the time when the new block was generated. According to the Ethereum agreement, when processing a new block, if the timestamp of this block is larger than the previous block and the difference between their timestamps is less than 900 seconds, then the timestamp of this new block is considered to be legal. The miner can freely adjust the timestamp of the new block within a specific range. Therefore, when a smart contract needs to use a timestamp as a trigger for important operations, there may be a vulnerability of timestamp dependency. The vulnerability of random number generation is similar. When a smart contract needs to use the index number, timestamp or block hash value as the seed generated by the random number, the vulnerability will be generated. Therefore, if any operation uses the number, timestamp, or blockhash instruction during execution, and the corresponding function is payable, we considered it to a dangerous operation.

Dangerous Delegatecall. The purpose of the delegatecall() is to implement a library call, which means the contract running environment is the caller's when calling from other contracts. Therefore, when the parameter of the delegatecall() is set to msg.data, there might be a dangerous operation. The call address and content of the delegatecall() are user-controllable, in which the attacker can make the victim contract calling any function it provides.

Exception Disorder. Exception disorder is caused by inconsistencies in exception handling between call contracts. When a contract calls another contract in a different way, if there is no consistent way to handle the exception, the calling contract may not be able to get the exception information in the called contract. For example, supposed a given chain of calls (a() calls b(), b() calls c(),...) until the exception is thrown, the following two situations will happen. First, each of these calls is a direct call to the contract function, and after an exception occurs, all transactions will be restored (including ether transfer); Second, at least one of them is called by call(), delegatecall() or send() or a low-level calling method, the transaction's rollback will only stop at the calling function and return false. At this point, some transactions are not restored and the thrown exception is not propagated. This inconsistency in exception handling will cause the called contract to ignore errors that occurred during execution. Therefore, we consider the case where the exception is not correctly propagated back to the root call as a dangerous operation.

Freezing Ether. If the smart contract used for the transferred business does not implement a transfer function by itself but relies solely on the code of other contracts, it may suffer by the vulnerability of freezing ether. When the contract provided transfer function performs a suicide or self-destruct operation, the ether of victim contract is possible frozen because of the failure of transfer function, especially those who call others via

delegatecall(). Therefore, when the contract corresponding function is payable and delegatecall() is used and the contract itself doesn't include transmit/send/call/suicide function, it is regarded as a dangerous operation.

Integer Overflow. The smart contract is written in solidity. Each data type has its own legal value range. If the developer does not consider their value range, there may be an overflow and a downflow error. For example, the value of the data type uint8 ranges from 0 to 255, and if the developer attempts to store 256 into uint8, the result will be 0, that is, an overflow occurs. In this paper, we will instrument in arithmetic instruction such as add, addmod, mul, mulmod, in order to identify the overflow.

Other types of vulnerabilities, such as short address attacks are mainly due to the fact that solidity features, so the algorithm in this paper does not deal with this vulnerabilities.

2.4 Genetic algorithm

For the coverage data classified by taint analysis, this paper proposes the genetic algorithm for smart contract to optimize fuzzing input. By continuously selecting individuals with high adaptability, the code coverage of smart contract fuzzing improves.

Genetic Algorithm (GA) is an optimized method that mimics the mechanism of biogenetics and evolution. It introduces behaviors similar to biogenetics such as cross-recombination, mutation, selection, and elimination in biology into the improvement process of algorithmic resolving problems. The genetic algorithm retains the locally optimal individual while at the same time obtaining better ones through the individual's genetic manipulation to the globally optimum result. Combing with the application scenario of smart contract [Liu, Yang, Zhang et al. (2017)], the specific process structure is shown as follows:

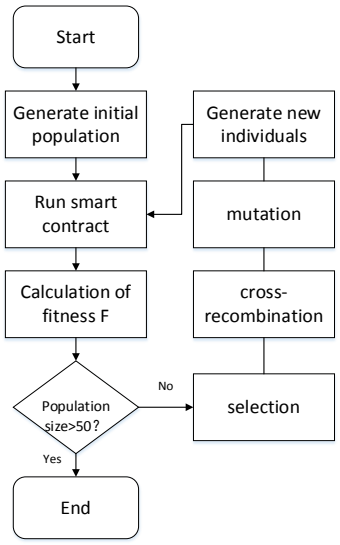

Figure 3: Genetic algorithm for smart contracts

First, the coverage data obtained from the classification of taint analysis (Section 2.2) is

used to generate the initial population. Here the initial population is the raw input data and its boundary values.

Second, the smart contract is tested with the current population, the state during the running of the program is recorded, and the fitness F of the individual is calculated.

In order to improve the coverage as much as possible, we consider the code coverage of the current input and the number of newly discovered code blocks while defining the fitness function of the genetic algorithm. The main function of the smart contract is for digital currency trading, so the functions in the contract are generally not complicated. Therefore we use the instruction *jumpi* as a sign of the smart contract code block. Formula is as follows:

$$F(x_i) = \frac{m_i}{M} + n_i \qquad (i = 1,2,\ldots) \tag{1}$$

In the formula, x_i is the i-th input, m_i is the number of *jumpi* covered by the i-th input data, M is the number of *jumpi* that have been found so far, and n_i is the number of newly discovered *jumpi* of the i-th input. The fitness function consists of the discovered code block coverage and the newly discovered code block. As M continues to grow, the contribution of code block coverage to the fitness function is decreasing, which is consistent with the continuous discovery with fuzzing, which means the priority of the new code block is increased.

The specific process is to traverse each individual of the contemporary population as input, record the code block of its execution, and add the newly discovered code block to the code-block-set. And then we call the fitness function for each individual to obtain the corresponding input's fitness.

Third, the initial data is going to be coded, and the individual with high fitness is selected to generate a new one.

The coding method of the genetic algorithm for smart contract is mainly divided into numerical type and non-numeric type, and the two types of parameters are encoded separately.

(1) Numerical data coding. In solidity, numerical data includes integers (int/uint) and bytes arrays, etc. The numerical data is binary coded, and the length is generated according to the input value in the requirement specification, for example, the number between -128 and 127 is represented by an 8-bit binary string.

(2) Non-numeric data coding. Non-numeric data mainly refers to such types as string, address, and bool. For bool inputs, it has only two states (true and false), which means that it does not need to be encoded. For the types of string and address, ASCII code of each character is converted to a binary code, which is then concatenated in alphabetical order as the encoding of the string.

In order to generate new individuals, two individuals are selected as parents according to their selection probability. The probability formula for selecting individuals is as follows:

$$P(x_i) = \frac{F(x_i)}{\sum F(x_i)} \qquad (i = 1,2,\ldots) \tag{2}$$

According to the probability of uniform distribution, one of the individual genes is selected as the exchange point, and then the genes of the two choices are exchanged to obtain the genes of filial generation. In order to improve the genetic diversity of the next generation population, there is a 20% probability for mutation among which $\frac{l}{20}$ bit (l is the data length, and the calculation result is rounded down), 10% probability for mutation among which $\frac{l}{10}$ bit, 5% probability for mutation among which $\frac{l}{5}$ bit. Once for non-fixed length input, set data length l to r (r is a random number), after the completion of process, the new gene will be added to the next generation of the population until the number of individuals reaches the standard of specified number.

Individuals with higher fitness will be affected by the crossover and mutation in the process of evolution, because of disadvantages in genetic algorithms. To solve this problem, we adopt the optimal preservation strategy to preserve those individuals of high fitness, and save the optimal solution before the selection operation, so that the algorithm can finally break through the local optimum to reach the global optimal value. The strategy of preservation is to let the optimal individual off being destroyed by crossover and mutation, which is an important condition for the improved adaptive genetic algorithm maintaining convergent.

After completing the genome generation of the filial generation, it is necessary to generate a certain number of inputs. In this paper, an extension stage is added before running smart contract. For each coverage data, the deterministic bit flipping is recorded as *x/y* variation, where *x* represents the bit length of the flipping, *y* represents the step length of the flipping operation. 1/1 means that each bit of the field is flipped one by one to generate a new gene. 1/1, 2/1, 4/1, 8/8, 16/8, 32/8, these six flip operations are defined in this paper. Bit flipping is equivalent to genetic algorithm further expansion of the population generated by mutation and adding new genes to the genome of the current population can more effectively discover new code execution paths.

After completing the genetic selection, mutation, and deterministic expansion, the input data set of the population is determined, and the next generation selection process can be performed.

Fourth, the new individual is decoded, and the smart contract runs with those new inputs. At the same time, the population is evaluated. If the algorithm is met with the standard of shutting down, the process is terminated. If not, the third step is repeated. In order to reduce the time of screening input data, and ensure the diversity of the population, this paper chooses the stopping condition as the number of individuals in the population is equal to 50.

3 Analysis on experiments and performance

3.1 System design

This method is based on the structure of the Ethereum blockchain to optimize, and experiments are carried out according to the following system diagram to verify the proposed fuzzing framework of smart contract (this method is hereinafter referred to as

T-framework).

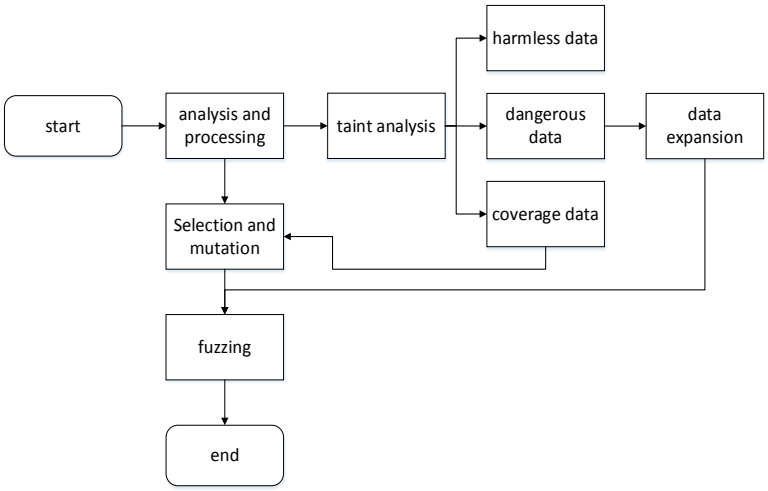

Figure 4: System diagram of T-framework

Hardware performance, system environment, and software configuration all have an impact on the results when performing fuzzing. And for this test, it is taken under the Windows 7 Professional Service Pack 1 64 bit, with a processor of Intel(R) Core(TM) i5-3337U 1.80 GHZ, with RAM of 4.00 GB.

The analysis processing module mainly performs static analysis on the smart contract, and generates raw input according to the function and data type in the ABI. If there is data with non-fixed length, a positive integer is randomly generated, which will be produced as its certain length. Meanwhile, the basic code block of the target smart contract are identified by the compilation tools to prepare for the calculation of the fitness function.

The taint analysis module is based on the taint analysis in Section 2.2 and the dangerous operation defined in Section 2.3, and it is in charge of marking, tracking and detecting the raw input data. Harmless data, coverage data, and dangerous data are also classified in this module.

The selection and mutation module takes the raw coverage input data obtained by the taint analysis and its boundary value as the initial population of the genetic algorithm, then take selective mutation to the data according to the method of Section 2.4 until the whole process stops.

The data expansion module is mainly for dangerous data, which will expand its boundary value and its bit flipping. It will be divided into two stages: the first stage is the same as the Section 2.4 x/y variation, and the second stage is a boundary value assignment for the dangerous type input data. Different boundary values are defined for the input data with length of 1, 2, and 4B, respectively. For example, the field boundary value of 1B includes -128, -1, 0, 1, 64, 127.

The fuzzing module is executed after the input data is generated. This module borrows the method of the ContractFuzzer to call the smart contract in the test for two modes: one is with ether transfer and one is not with. And we start the HTTP server to collect and

analyze storage space for fuzzing too.

In order to test the coverage rate and other performance of the proposed method, we select vulnerability detection tools ContractFuzzer and Oyente for comparison, both two with high acceptance. ContractFuzzer is the first fuzzing framework for smart contract vulnerabilities detection based on Ethereum. It supports the detection of seven vulnerabilities: gasless send, exception disorder, reentrancy, timestamp dependency, random number generation, dangerous delegatecall, and ether freezing. Oyente is one of the most widely used smart contract detection tools, which is a static analysis tool based on symbolic execution. It supports vulnerabilities detection such as reentrancy, timestamp dependency, and exception disorder. Among the types of vulnerabilities that support, Oyente supports fewer than T-frameworks and ContractFuzzer, while T-frameworks add a function as the detection of integer overflow compared to ContractFuzzer.

3.2 Analysis of performance

This article selects 150 smart contracts on Etherscan as test samples to analyze the performance of T-framework.

Considering the differences in the types of vulnerabilities supported by the three tools, we only compare the detection effects of the three vulnerabilities of reentrancy, timestamp dependency, and exception disorder. The results are shown in the following table, which is analyzed by TP (True Positive), FP (False Positive), and Precision.

Table 2: Performance comparison among T-frameworks, ContractFuzzer, and Oyente

vulnerabilities	Tools	TP	FP	Precision
Reentrancy	T-framework	4	0	100%
	ContractFuzzer	3	1	75%
	Oyente	3	1	75%
Timestamp Dependency	T framewoek	13	1	92.9%
	ContractFuzzer	13	1	92,9%
	Oyente	10	7	58.8%
Exception Disorder	T-framework	5	0	100%
	ContractFuzzer	5	0	100%
	Oyente	5	4	55.6%

The T-framework detection effect are almost the same as ContractFuzzer. However, the ContractFuzzer detection principle for the reentrancy is detected by the call of attacker contract, so the FP are slightly higher than those of T-framework. While comparing with Oyente, the overall detection effect of the T-framework is better than that of Oyente.

Secondly, we also make statistics on the average detection time of single contract of the above tools. Oyente is about 28.4 s, T-framework is about 47.6 s, and ContractFuzzer is about 56.1 s. Although the detection time of T-framework is slightly longer than that of

Oyente, it is still in an acceptable range. Compared with ContractFuzzer, the detection efficiency of T-framework has been greatly improved. This is because of the introduction of taint analysis and genetic algorithm, which makes the generation of fuzzing data more oriented, rather than completely random variation.

3.3 Analysis of performance

In the coverage analysis, since oyente is a static analysis tool, we only choose ContractFuzzer as the comparison. At the same time, due to the fact that the code coverage will rise quickly when the simple contract is tested, it can not prove the improvement of the coverage.

Here we select a more complex smart contract for multi-party voting options as a sample. Through static analysis, the smart contract has 15 callable functions, and the number of *jumpi* is 44. In the same machine, T-framework and ContractFuzzer are used to test the above smart contract respectively. In the time t=100 s, we take each wheel generation in the T-framework to complete the fuzzing as the unit time, and record it as T_g, and then count the generation code coverage every T_g. The results are shown in Fig. 5, where the abscissa represents the detection time and the ordinate represents the number of *jumpi* covered.

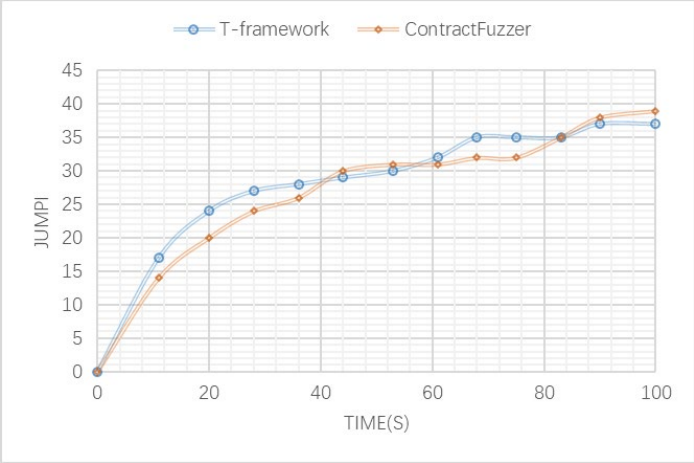

Figure 5: Coverage comparison between T-frameworks and ContractFuzzer

It can be seen from the above figure that when $t<5T_g$, the number of *jumpi* executed by T-framework is more than that of ContractFuzzer. That is, the code coverage of T-framework in the early stage of detection is better than that of ContractFuzzer. This is because the T-framework selects the data through the taint analysis, and on this basis, the genetic algorithm is used to provide guidance for the generation of fuzzing input. However, when $7T_g>t>5T_g$, the T-framework slightly meets the bottleneck of coverage growth, which may be that some individuals with excellent genes are lost in a more complex function. This also leads to the code coverage not being higher than ContractFuzzer in this period of time. When $10T_g>t>7T_g$, the code coverage of *T*-framework is still better than that of ContractFuzzer. But it should note that when $t=8T_g$, the code coverage of T-framework encounters a long bottleneck period, which is also due

to the disadvantage of genetic algorithm, that is, the loss of some genes leads to the local optimum rather than the global optimum. Therefore, when $t>10T_g$, the code coverage of ContractFuzzer will be slightly higher than that of T-frame. At this time, T-framework may produce excellent offspring with extremely small probability to form a small increase, but its overall trend has been relatively flat. If we need to continue to increase code coverage, it will take a very long time. Generally speaking, for a smart contract, the detection time range we can accept is about 60 s.The code coverage of T-framework in that short-term range is expected to be higher than that of ContractFuzzer, which also roughly achieves our goal of balancing code coverage and detection efficiency.

5 Conclusion

In terms of shortcomings of the detection tools of current smart contract vulnerability and the architecture of Ethereum, this paper proposes a smart contract fuzzing method assisted by taint analysis and genetic algorithm. The main contributions of this paper are stated as follows:

(1) For the redundancy problem of input data of fuzzing, we research and analyze the EVM instruction execution process and method of instrumentation, and propose a method of taint analysis for Ethereum smart contract, which optimizes the generation of fuzzing input, thus improving the efficiency of smart contract fuzzing.

(2) In the view of the definition and identification of dangerous operations, the common vulnerabilities of smart contracts are investigated, and the dangerous operations of smart contract execution are analyzed and summarized in turn, which supports the classification of input data in smart contract by taint analysis.

(3) Aiming at the problem of relatively low coverage of smart contract fuzzing, a genetic algorithm adapted to smart contract is designed. The algorithm selectively optimizes the coverage data, which improves the code coverage of the smart contract fuzzing.

Acknowledgment: Thanks for Bichen Che, Yu Yang, and Zichuan Guo, who give a lot of suggestions and contribute to this article.

Funding Statement: This work is supported by the National Key R & D Program of China (2017YFB0802703), Major Scientific and Technological Special Project of Guizhou Province (20183001), Open Foundation of Guizhou Provincial Key VOLUME XX, 2019 Laboratory of Public Big Data (2018BDKFJJ014), Open Foundation of Guizhou Provincial Key Laboratory of Public Big Data (2018BDKFJJ019) and Open Foundation of Guizhou Provincial Key Laboratory of Public Big Data (2018BDKFJJ022).

Conflicts of Interest: We have no conflicts of interest to report regarding the present study.

References

Bekrar, S.; Bekrar, C.; Groz, R.; Mounier, L. (2012): A taint based approach for smart fuzzing. *IEEE Fifth International Conference on Software Testing, Verification and Validation*, vol. 1, no.1, pp. 818-825.

Chris, C. (2019): A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

Grishchenko, I.; Maffei, M.; Schneidewind, C. (2018): A semantic framework for the security analysis of ethereum smart contracts. *Principles of Security and Trust*, vol. 1, no. 1, pp. 243-269.

Jiang, B.; Li, Y.; Chan, W. K. (2018): Contractfuzzer: fuzzing smart contracts for vulnerability detection. *ACM International Conference on Automated Software Engineering*, vol. 1, no. 1, pp. 259-268.

Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. (2018): ZEUS: Analyzing safety of smart contracts. http://pages.cpsc.ucalgary.ca/joel.reardon/blockchain/readings/ndss2018_09-1_Kalra_paper.pdf

Liu, Y.; Yang, Y. H.; Zhang, C. R.; Wang, W. (2017): A novel method for fuzzing test cases generating based on genetic algorithm. *Acta Electronica Sinica*, vol. 45, no. 3, pp. 552-556.

Luu, L.; Chu, D. H.; Olickel, H. (2016): Making smart contracts smarter. *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, vol. 1, no. 1, pp. 254-269.

Michael, S.; Adam, G.; Pedram, A. (2007): *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

Nakamoto, S. (2009): Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf