Security Analysis of Blockchain Smart Contract: Taking Reentrancy Vulnerability as an Example

Mingtao Ji[1], GuangJun Liang[1,2](✉), Meng Li[1], Haoyan Zhang[1], and Jiacheng He[1]

[1] Department of Computer Information and Cyberspace Security, Jiangsu Police Institute, Nanjing, China

[2] National and Local Joint Engineering Laboratory of Radio Frequency Integration and Micro Assembly Technology, Nanjing University of Posts and Telecommunications, Nanjing, China

Abstract. As the blockchain enters the 2.0 era, the smart contract which is based on the blockchain platform has gradually entered people's field of vision. By its transparency, non-tampering, independence from third-party arbitration, and trustlessness, it is widely applied in equity crowdfunding, games, insurance, particularly the Internet of Things. However, the attack on the TheDAO smart contract alert public awareness of the security of a smart contract. The essence of the smart contract is an electronic contract written in code. Due to reasons like lacking standard libraries for its programming language, several loopholes will inevitably appear in the code. Once they are found by attackers, the interests of the main body using smart contracts will be damaged. There are many types of vulnerabilities in smart contracts, such as reentrancy, short address attacks, and timestamp dependence. This article mainly focuses on re-entry vulnerabilities as the research object and analyzes the principle of re-entry vulnerabilities. Featuring immutable after being chained, the smart contract must be checked before it is chained to make up for the vulnerability. This paper provides a detection method based on symbolic execution to detect reentrancy vulnerabilities. We hope to strengthen people's security awareness of smart contracts and boost the research of smart contracts in terms of security by our study of the smart contract reentrancy vulnerability in this article. Promote the research and development of smart contracts.

Keywords: Smart contract · Blockchain · Reentrancy vulnerability · Fallback function · Symbolic execution

1 Introduction

The publishment of an article titled "Bitcoin: A Peer-to-Peer Electronic Cash System"[1] by a mysterious person named Satoshi Nakamoto as well as the open-source of the Bitcoin system [2] marked the great invention of blockchain. The advantages of blockchain, including tamper resistance, transparency and credibility, and protection of privacy, meet people's needs and attract people's attention. With the development of informatization, blockchain has broken through the limitations of digital currency applications, being utilized in digital assets and smart contracts [3]. Moreover, smart contracts which run on the

X. Sun et al. (Eds.): ICAIS 2021, CCIS 1424, pp. 492–501, 2021.
https://doi.org/10.1007/978-3-030-78621-2_41

blockchain are widely used in the fields of finance, energy system, deposit certificates, and digital copyrights.

However, every coin has two sides. Though the literature [4–6] all recognized the advantages of blockchain, they raised questions about the security of the blockchain. Literature [4] investigated the challenges faced by blockchain in terms of technology, risks, and security. Literature [5] pointed out that the attack on the blockchain is the vulnerability itself in the context of extensive application. Literature [6] proposed some vulnerabilities majorly consisting of design flaws, code-level implementation flaws, ecosystem problems (such as wallets), and 51% attacks. The following two cases sufficiently prove the existence of security problems.

Blockchain attacks are all exploiting code vulnerabilities in smart contracts on the blockchain. This triggered our thinking about the security of smart contracts. The literature [7–10] all analyzed the loopholes of smart contracts running on the blockchain. Literature [7] pointed out that because smart contracts cannot be modified or updated once they are on the chain, the best way to prevent vulnerabilities in a smart contract is to check the contract code before it is deployed on the blockchain. Literature [8] comprehensively categorizes smart contract security issues and uses secure code analysis tools that identify known vulnerabilities to thoroughly review known vulnerabilities. The literature [10] specifically introduces the loopholes of the Ethereum smart contract and the corresponding defense mechanism. Literature [9] introduced a general technique for building core functional models for model checking to help alleviate the security issues in smart contract development.

2 Smart Contract Introduction

2.1 What is a Smart Contract

The definition: "A smart contract is a set of commitments defined in digital form, including a contract that participants can perform these commitments on its agreement [11]," is the earliest concept of the smart contract which was proposed by scholar Nick Szabo in 1994. A smart contract is like a vending machine. Customers need to select a product and complete the payment. Afterward, the vending machine will automatically spit out the product.

Partially, smart contracts have the same attributes as real-life contracts that both parties need to sign a certain agreement in light of law whereas the difference lies in that there is no possibility of the bargain, discount, or something sophisticated during the transactions in the smart contract. After all, the only goal of codes is to execute themselves strictly under established rules and output.

2.2 Smart Contracts in Various Blockchain Platforms

The development of the blockchain can be divided into three stages, namely the 1.0, 2.0, and 3.0 era [12]. Each period has a distinctively typical blockchain platform on which smart contracts are not rare to be seen. Such as the Bitcoin system in the era of blockchain 1.0, Ethereum in the era of blockchain 2.0, Hyperledger in the era of blockchain 3.0. Scripts are simple, stack-based execution languages that are processed from left to right.

A blockchain-based smart contract is comprised of a transaction processing mechanism, a data storage mechanism, and a complete state machine for receiving and processing various conditions.

The principle of the smart contract running on the blockchain system is shown in Fig. 1. Each node has its smart contract that contains the balance of the account and the information of the blockchain where it is located. When the preset trigger conditions in the smart contract are met, the smart contract accesses the corresponding data for calculations according to the preset response rules. Then it saves up the results on the blockchain. In other words, after the smart contract runs, the smart contract account which contains account balance, storage, and other content is automatically generated and store up on the blockchain [13]. Subsequently, when executing the contract code (also called smart contract call) in the virtual machine, as soon as every node reaches a consensus on the ultimate execution result, the state of the smart contract on the blockchain will be updated accordingly or even create a new block (see the dotted box in Fig. 1).

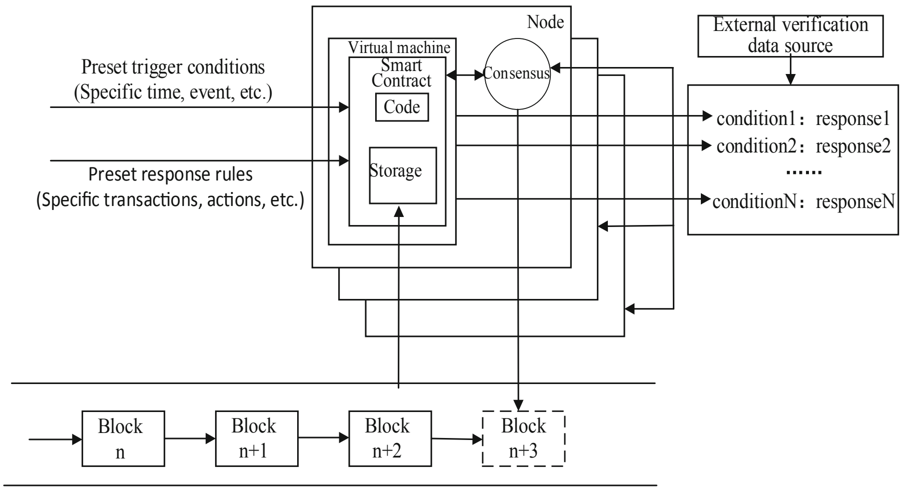

Fig. 1. The principle of running smart contracts on the blockchain system

3 Smart Contract Vulnerability Analysis: Reentrancy Vulnerability

The prosperity of smart contracts drives the increasing number of smart contracts. Additionally, the popularity of decentralized applications prompts digital assets involved in smart contracts to boom significantly. Due to the difficulty of patching smart contracts after they are on the chain and the lack of assessment standards for ensuring smart contract quality, they have become the target of hacker attacks [14]. This article mainly studies reentrancy vulnerabilities.

3.1 Overview of Reentrancy Vulnerabilities

Reentrancy vulnerability emerges in a virtual machine. One of the characteristics of Ethereum smart contracts is the ability to call and utilize the codes of other external contracts. Besides, it is normal for contracts to deal with Ether in the way that the Ether would be sent to various external user addresses when being processed. Both transfers and calls to external contracts require the contract to submit external calls. However, it is these external calls that are easily hijacked by attackers. These attacks and improper operations are carried out by attacking the contract itself through the fallback function or callback. Similar to the reentrant phenomenon that occurs when the operating system is interrupted during process scheduling, reentrancy is usually manifested as multiple calls to a function at the same time. The malicious contract calls the attacked function multiple times before calling other functions to complete. The code execution is re-entered in the attacked contract to realize the attack, which may result in huge damage [15].

In general, the key point of the attacker's reentrancy attack lies in the fallback function on account of the malicious exploitation of the fallback function of the smart contract for reentrancy vulnerabilities.

3.2 Functions Involved in Reentrancy Vulnerabilities

Fallback Function. The fallback function is a special function in a smart contract in that it has no name, no parameters, and no return value. Therefore, it frequently calls other functions. By and large, the fallback function will be called when there is no available function for the contract to call or the call does not contain any data.

Withdraw Function. To allow users to perform withdrawal operations(namely convert the tokens in the contract system into the ether and withdraw), withdraw function often exists in contracts such as wallets and decentralized transactions. As a result, this has become the focus of reentrancy vulnerability attacks [16].

Other Functions. In addition to the above two key functions that lead to reentrancy vulnerabilities in the smart contract code, other functions ensure the normal operation of the code. Such as call, transfer, send, and balance functions. The call function is also one of the reasons for the reentrancy vulnerability.When using $<$ address $>$.gas().call.value()() to transfer, all available gas can be called, which cannot effectively prevent reentrancy.

3.3 Principles of Reentrancy Vulnerability Attack

When attackers attack smart contracts with reentrant vulnerabilities, they exploit the fallback function of the attacker's contract and the withdraw function of the attacked contract. During the transferring process, the attackers will use the reentrance vulnerability to "modify the Storage variable and transfer" the contract which is supposed to be an atomic [17] transaction. Use the sequence of a first transfer before modifying Storage variables. If the target of the transfer is a contract with a malicious fallback function, it may be called by the malicious contract to the victim contract, thereby destroying the atomicity of the operation and bypassing the check to repeatedly obtain the transfer revenue [18].

Take the Bank contract [19] as an example:

As is shown in Fig. 2 Bank contract part code, balances[msg. sender] represent the amount of Ether balances the user has. The purpose of the withdraw function is to allow users to withdraw assets through the receiver.call.value(amount). The call function is a function for sending ether in a smart contract. When the user calls this function, the contract first checks whether the user balance is bigger than the number of funds withdrawn. If the check is passed, the requested asset will be transferred to the user in the form of Ether, and the corresponding balance will be deducted from the user account.

Owing to the operating mechanism of the Ethereum smart contract that if the address receiving the transfer is a contract address, the fallback function of that address will be triggered, this mechanism may be utilized by malicious attackers to launch reentrancy attacks. An attack is an attack contract, and its code is shown in Fig. 3.

The attacker only needs to call the withdraw function of the victim contract through the step2(uint256 amount) public function. When the receiver. call.value(amount)() statement in the withdraw function of the victim contract is executed, it will trigger the fallback function in the attack contract. In this function, the call to the victim contract withdrawal is initiated again, and the recursive call will be repeatedly initiated before the user's balance is reduced until the Gas is exhausted, thereby continuously stealing the ether in the victim contract.

The entire attack process is shown in Fig. 4. The attacker triggers the fallback function when transferring money to the vulnerable contract through the attack function, and then uses the fallback function to repeatedly call the withdraw function to construct a loop between steps 2 and 3 to continuously steal the ether in the victim contract. In this process, the vulnerable contract continues to transfer money to the attacker contract until the end of the cycle.

```
  contract Bank {
  ……
  function withdraw(address receiver, uint256 amount) public{
  require(balances[msg.sender] > amount);
  require(address(this).balance > amount);
  receiver.call.value(amount)();
      // When using call.value()() for ether transfer, there is no gas limit
  balances[msg.sender] -= amount;
  }
  function balanceOf(address addr) public view returns (uint256) {
  return balances[addr];
  }
}
```

Fig. 2. Bank contract part code

```
        contract Attack {
          ......
           function step2(uint256 amount) public{
              victim.call(bytes4(keccak256("withdraw(address,uint256)")),
  this,amount);}
          ......
           function () public payable {
              if (msg.sender == victim) {
                 victim.call(bytes4(keccak256("withdraw(address,uint256)")),
  this,msg.value);}
           }
        }
```

Fig. 3. Bank contract attack part of the code

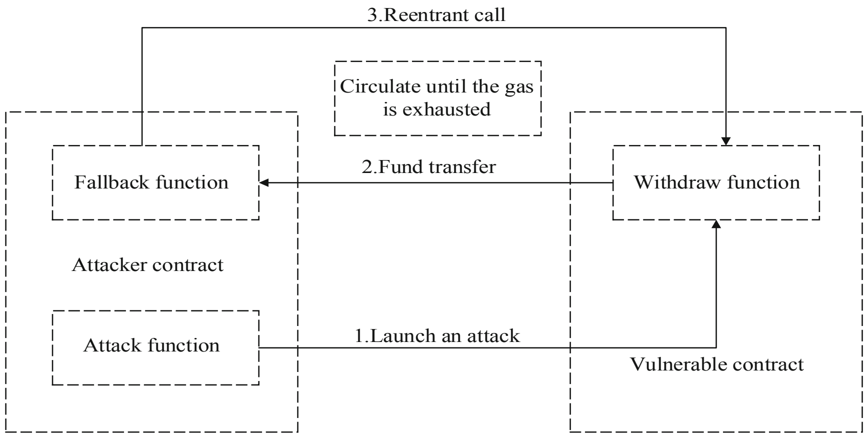

Fig. 4. Brief flowchart of reentrancy attack

Using reentrant vulnerabilities to attack contracts is mainly to fully call all available gas through the gas mechanism of the call instruction of the Ethereum virtual machine, as well as features such as the fallback function mechanism and recursive access that allows low-level calls.

4 Smart Contract Security Vulnerability Detection Method

In fact, by analyzing a large number of cases that smart contracts being attacked cases, we can easily conclude that the vulnerabilities of smart contracts are mainly in the code. Lacking standard libraries which could be commonly found in high-level programming languages, it is difficult for all developers to program smart contract, reducing the security of the code [20]. Furthermore, the code of the smart contract cannot be changed once it is on the chain. In this case, to prevent the smart contract from being attacked, we need to check the code before the contract code is on the chain.

At present, the main detection methods include five methods: formal verification, symbolic execution, static analysis, dynamic strain analysis, and fuzz testing [21]. This article centers on symbolic execution-based methods to detect code reentrancy vulnerabilities.

4.1 Symbolic Execution Principle

Symbolic execution is a promising technique adopted widely in smart contract analyzers. It abstracts variables into symbols, which are then used as program input. The symbolic execution method explores all possible program execution paths through the input of abstract symbols and uses a constraint solver to calculate a concrete input (test case) for each feasible path [22, 23].

Time witnessed the evolution of symbolic execution from static symbolic execution to dynamic symbolic execution. The core idea of static symbolic execution is to use abstract symbol values to replace the specific variable values of the program, and to represent the actual program input with symbolic input. However, this method is only suitable for relatively simple programs. Moreover, this execution method cannot obtain system calls or third-party library call functions [22].

Therefore, Godefroid [24, 25] proposed a dynamic symbolic execution method. Combining concrete execution with symbolic execution and using concrete values instead of symbolic values as the input of the program, the analysis accuracy has been widely applied in recent years for higher accuracy and plain feasibility.

Static Symbolic Execution. In the process of symbolic execution, each branch statement is searched when a program branch statement is encountered. At the same time, the symbolic execution analyzer will add the branch condition of the branch statement to the constraint condition set of the current path. To determine whether the target area code is reachable, a constraint solver [26] needs to be used to verify the solvability of all constraints. If the constraint solver successfully solves the value, the path is reachable; otherwise, the path is unreachable, thus ending the further analysis of the path [27].

Static symbol execution is similar to the select the statement in C language, but there are also differences. As shown in Fig. 5, when the C language program runs to the if statement, if the condition of the if statement is met, the else statement no longer runs . In static symbol execution, the if…else statement is regarded as two branches to analyze. As soon as the two variables reach the numerator along the established path of the program, we assign the value of $A + 1 - B$ to A, namely the statement $A = A + 1 - B$. On condition that the eighth line of code is the correct code and both branches are executable, then at the if statement, $A + 1 - B < 0$; at the else statement $A + 1 - B \geq 0$ [27, 28]. To execute the path of the error code after the else statement, it needs to conform to conditional statement $A + 1 - B \leq 0$ [26].

Dynamic Symbolic Execution. Supposing that the code is replaced with $x = x*x - 2*y$ as others remaining unchanged. The execution steps are shown in Fig. 6. The actual value is randomly determined as the test seed input, and the path to execute the

```
int fun(int x,int y)
{
    x++;
    x=x-y;
    if(x<0)
        x++;
    else
        //some error
    return x;
}
```

Fig. 5. C language code example

target program is the path corresponding to the actual value. The execution path also maintains the symbolic execution information of the program, that is, the symbol status corresponding to the path and the constraint conditions representing the path, and the constraints are collected at the same time. In the execution process, the corresponding constraint conditions of the branch statement of the program that have not been explored are obtained by inverting the current branch condition. Then, determine the satisfiability of the branch. If it is unsatisfied, it cannot be executed, and if it is satisfied, the calculated value will be saved as the data for the subsequent generation of test cases [29]. Finally, the loop is repeated until the traversal is completed.

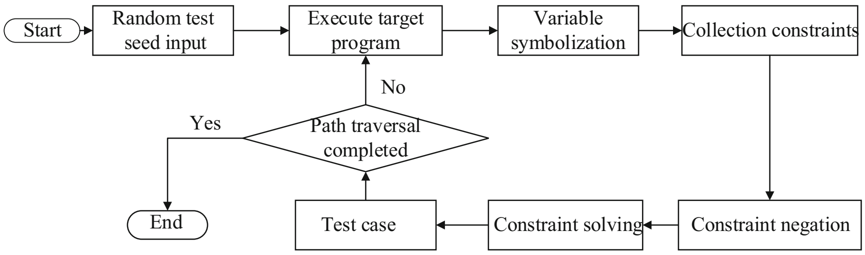

Fig. 6. Dynamic symbolic execution steps [29]

4.2 The Basic Process of Using Symbolic Execution to Detect Contract Vulnerabilities [29]

Using the principle of dynamic symbol detection to detect smart contract vulnerabilities, the specific steps are shown in Table 1.

Table 1. Basic steps to detect contract vulnerabilities using symbolic execution.

Basic steps to detect contract vulnerabilities using symbolic execution
1. Use the solc compiler to compile the contract source code to generate assembly code, which contains deployment code, runtime code, and aux data
2. Use the solc compiler to decompile the runtime code to generate the ethereum contract bytecode
3. Construct a control flow graph through the Ethereum contract bytecode
4. Generate test data randomly, traverse the reachable path of the control flow graph, and collect path constraints
5. Use the constraint solver to solve the path constraints and generate test cases

5 Conclusion

As a vital feature of the blockchain 2.0 era, smart contracts play an irreplaceable role on the Ethereum platform. Since the occurrence of the TheDAO incident, the security of smart contracts has attracted widespread attention. This article analyzes the reentrant vulnerability at the Ethereum virtual machine level and explains how TheDAO was attacked. At the end of this article, a detection method based on symbolic execution is proposed to detect the contract before the chain to avoid loopholes.

Acknowledgments. This work is supported by Open project of National and Local Joint Engineering Laboratory of Radio Frequency Integration and Micro-assembly Technology, Nanjing University of Posts and Telecommunications (KFJJ20200201), Talent Introduction Project of Jiangsu Police Institute (JSPI19GKZL407), Scientific Research Project of Jiangsu Police Institute (2110120030) and Jiangsu Province University Student Innovation Training Program Project (202010329034Y).

References

1. Nakamoto, S.F.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf
2. Qing, B., et al.: Bitcoin and legal digital currency. J. Cryptologic Res. **4**(2), 176–186 (2017)
3. Zh, J.M., Yang, F., Fu, F.: Research progress of blockchain applications. Sci. Technol. Rev. **55**(9), 70–76 (2017)
4. Kadena, E., Holicza, P.:Security issues in the blockchain (ed) world. p. 216. In: CINTI (2018)
5. Jonathan, K., Sari, A.K.: Security issues and vulnerabilities on a blockchain system: a review. In: ISRITI, Yogyakarta, pp. 228–232, Indonesia (2019)
6. Keenan, T.P.: Alice in blockchains: surprising security pitfalls in PoW and PoS blockchain systems. In: PST, pp. 400–4002, Calgary, AB (2017)
7. Sayeed, S., Marco-Gisbert, H., Caira, T.: Smart contract: attacks and protections. In: IEEE Access **8**, 24416–24427 (2020)
8. Dika, A., Nowostawski, M.: Security vulnerabilities in ethereum smart contracts. In: 2018 IEEE International Conference on Internet of Things, Physical and Social Computing and IEEE Smart Data, pp. 955–962, Halifax, NS, Canada (2018)

9. Kongmanee, J., Kijsanayothin, P., Hewett, R.: Securing smart contracts in blockchain. In: ASEW, pp. 69–76, San Diego, CA, USA (2019)
10. He, D., Deng, Z., Zhang, Y., Chan, S., Cheng, Y., Guizani, N.: Smart contract vulnerability analysis and security audit. IEEE Netw. **34**(5), 276–282 (2020)
11. http://virtualschool.edu/mon/Economics/SmartContracts.html. Accessed 21 Nov 2020
12. Cao, B., Lin, L., Li, Y., Liu, Y.X., Xiong, W., Gao, F.F.: Overview of blockchain research. J. Chongqing Univ. Posts Telecommun. (Nat.Sci. Ed.) **6**, 1–14(2020)
13. Han, X., Liu, Y.M.: Research on consensus mechanism in blockchain technology. Netinfo Secur. **9**, 147–152 (2017)
14. Liao, J.W., Tsai, T.T., He, C.K., Tien, C.W.: SoliAudit: smart contract vulnerability assessment based on machine learning and fuzz testing. In: IOTSMS, pp. 1959–1966, Granada, Spain (2019)
15. Qiu, X.X., Ma, Z.F., Xu, M.K.: Analysis and countermeasures of security vulnerabilities of ethereum smart contract. In: Information Security and Communications Privacy, pp. 44–53 (2019)
16. Wang, Y.Z., Chen, J.L., Wang, X., He, Z.S.: Smart contract security analysis and audit guide. Publishing House of Electronics Industry, no. 11, p. 142 (2019)
17. Luu, L., Olickel, H., et al.: Making smart contracts smarter. In: SIGSAC, pp. 254–269, Vienna Austria (2016)
18. Ni, Y.D., Zhang, C., Yin, T.T.: A review of research on smart contract security vulnerabilities. J. Cyber Secur. **34**, 78–99 (2020)
19. CSDN Homepage. https://blog.csdn.net/weixin_43405220/article/details/100553931. 10 Nov 2020
20. Huawei Blockchain Technology Development Team.: Blockchain technology and application. Tsinghua University Press (2019)
21. Zhong, Z.B., Wang, C.D., Cai, J.H.: An overview of the research status and detection methods of smart contract security. In: Information Security and Communications Privacy, pp. 93–105 (2020)
22. Yang, K.: Automated security audit of smart contract based on symbolic execution. M.S. dissertation, University of Electronic Science and Technology (2020)
23. Zhang, W., Banescu, S., Pasos, L., Stewart, S., Ganesh, V.: MPro: combining static and symbolic analysis for scalable testing of smart contract. In: ISSRE, pp. 456–462, Berlin, Germany (2019)
24. Godefroid, P., Klarlund, N., Dart, K.: Directed automated random testing. In: ACM, pp. 213–223 New York, NY, USA (2005)
25. Sen, K., Marinov, D., Cute, G.: A concolic unit testing engine for. In: ACM, pp. 263–272, New York, NY, USA (2005)
26. de Moura, L., Nachmanson, L., Wintersteiger, C.M.: The Z3 Theorem Prover (2018). https://github.com/Z3Prover/z3
27. Zhang, Y.W.: Dynamic symbolic execution constraint solving optimization design and realization, M.S. dissertation, Chongqing Universit (2017)
28. Tan, C.: Security detection system for ethereum smart contract code. M.S. dissertation, University of Electronic Science and Technology (2019)
29. Zhao, W., et al.: Smart contract vulnerability detection scheme based on symbolic execution. J. Comput. Appli. **40**(4)947–953(2020)