On the Detection Limitations of the Re-entrancy Attacks on Ethereum

Jialu Fu, Wenmao Liu, Chaoyu Zeng, and Wenfeng Huang

Abstract In recent years, the emergence of Ethereum has brought people a new way of life. Many users tend to deposit funds into different smart contracts, but the smart contracts are actually different computer programs, so there may be some bugs and vulnerabilities in the smart contract that cause economic losses or bring potential dangers. Since the infamous attack on the "TheDAO" smart contract in 2016 until now, re-entrancy attacks, as one of the main attack methods on Ethereum, have caused serious losses. In response to this problem, many works of literature have proposed off-chain auditing of undeployed smart contracts and on-chain detection of deployed smart contracts, but re-entrancy attacks still emerge endlessly. In this paper, we introduce the limitations of Ethereum re-entrancy attack detection from the causes of re-entrancy attacks, behavioral characteristics, and the shortcomings of existing detection methods. First, we analyze the causes of re-entrancy attacks based on the execution characteristics of smart contracts in actual transactions, and then propose two deficiencies in the run-time attack detection method. Secondly, we selected re-entrancy attack transactions that actually occurred and were officially reported, manually analyzed the smart contracts and call sequences involved in these transactions, summarized two key factors of re-entrancy attacks, and analyzed different re-entrancy attacks, summarizing their behavioral characteristics at the theoretical level.

Keywords Ethereum · Smart contracts · Re-entrancy · Attack behaviors analysis

J. Fu · W. Huang
Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 51006, China

W. Liu (✉)
NSFOCUS Technologies Group Co., Ltd., Beijing 100089, China
e-mail: liuwenmao@nsfocus.com

C. Zeng
National Computer System Engineering Research Institute of China, Beijing 102699, China

S. Li (ed.), *Computational and Experimental Simulations in Engineering*, Mechanisms and Machine Science 146, https://doi.org/10.1007/978-3-031-44947-5_5

1 Introduction

As a computer program, the smart contract runs on Ethereum, which is a bridge between the blockchain and users. Users use smart contracts to complete operations such as deposits and loans. After the smart contracts perform these operations, the final results are stored permanently in the blockchain.

As a way to directly operate user assets, smart contracts must provide users with sufficient security guarantees, but because smart contracts cannot be modified once they are deployed on the blockchain, and the characteristics of arbitrary calls between contracts have attracted many attackers to use these features steal funds from users or various projects.

Re-entrancy attack is one of many attack methods. Starting from theDAO in 2016, it has evolved into different types, such as Lendf.me, Rari Capital, etc., and each attack has caused huge economic losses.

In order to reduce the occurrence of re-entrancy attacks, many studies [1–10] have proposed different re-entrancy attack detection tools, some for the run-time detection of on-chain transactions, and some for off-chain audits of undeployed contracts. However, some of these tools use a certain attack event as a reference case to analyze the behavior of re-entrancy attacks and formulate a detection plan, which will lead to too strict detection conditions and cannot be applied to detect new reentrancy attacks; some tools do not distinguish between re-entrancy attacks and normal re-entrancy behaviors. Such detection rules are too broad and prone to misjudgment, which will bring unnecessary trouble to tool users to a certain extent.

Contributions. In this article, we first analyze the cause of the re-entrancy attack, and conclude that the re-entrancy attack is not only due to random calls between contracts, but because of the characteristics of smart contracts that make the victim contract call unknown contracts during execution. Secondly, from the perspective of the attacker, analyze the two key factors in the re-entrancy attack: how to select the attack target and how to operate the state in the victim's contract to successfully complete the transaction. Re-entrancy attacks are classified and covered in depth based on these two factors and the attack transactions that have occurred. Finally, we summarize and analyze the omissions in detection of existing runtime attack detection tools.

Here is a brief introduction to the structure of the paper. In Sect. 3, we introduce the concepts of re-entry and re-entrancy attacks, and analyze the causes of re-entrancy attacks; in Sect. 4, we classify re-entrancy attacks from two key points of attack behavior; in Sect. 5, we introduce several existing re-entrancy attack detection tools, and analyze possible deficiencies in these detection methods.

2 Background

Ethereum Virtual Machine. EVM [11, 12] is the environment in which the smart contract runs. When the smart contract is deployed on the blockchain, it will be compiled into a piece of bytecode. When an external user invokes a transaction, EVM will execute the smart contract according to the input data of the transaction and it will revert the transaction or put the execution result on the blockchain, and change the state stored on the Ethereum by executing the transaction. EVM is a stack-based state machine that operates according to a series of opcode instructions [13], such as ADD, CALL, MLOAD, SSTORE, etc.

Smart contract. The smart contract is a computer program written in a high-level language, which is used to implement various functions provided by Ethereum to users. When it is deployed on the blockchain, it cannot be changed, and it can only be executed when called by EOA (external owned account) or other smart contracts.

Transaction. A transaction invoked by an EOA is called an external transaction, and the callee can be other EOA or smart contracts, which can be transfer operations, contract creation operations, or smart contract calls on the blockchain; after the contract is deployed on the blockchain, it cannot be automatically executed at any time but requires an external transaction to call these contracts. When the contract called by the external transaction invokes other contracts on the blockchain, it is called an internal transaction.

State. The contract state is stored in the storage [14]. Once each contract is deployed, a part of the storage space will be allocated to permanently store the contract data. The data and changes will be recorded on the blockchain and stored in the form of key-value. EVM can use SLOAD and SSTORE read and write storage, however, they read and write data on the blockchain, so the cost of these two operations is relatively high. Generally, smart contracts store important data in storage, for example, token contracts store *user-user balances* in storage.

Message call. When an external call to other contracts from the running contract, the execution of the caller contract will be interrupted, and the EVM turns to execute the callee contract. After the execution of the callee contract, it will return to the caller contract and continue to execute until the execution ends. However, there is uncertainty in the call sequence among contracts, so predicting the call relationship between contracts in advance is difficult.

Call stack. The call stack is used to record the calls among all contracts in a transaction, so as to determine the caller and callee in each call. When a new contract is called, it will be pushed into the stack and popped out after the execution returns. If the same contract appears in the call sequence, it can also be distinguished by the call stack.

3 Re-entrancy Attack

3.1 The Problem of Re-entrancy

Due to the feature of allowing smart contracts to call each other in the internal transaction, if contract A is interrupted during execution, and a call to contract A appears again in the call stack after contract A is interrupted, then it is considered that there is a re-entrant call or normal re-entrancy behavior. Re-entrancy attacks occur in re-entrant calls. For example, an attacker can use re-entrant calls to interrupt the modification of certain states of the contract or successfully pass judgment on certain conditions, transfer the assets of the victim's contract, and obtain funds that are more than the value of the funds invested in the attack.

3.2 Causes of Re-entrancy Attacks

In order to increase the reuse of contract codes and separate functions, different contracts need to call each other to provide users with a complete service. Therefore, during the execution of functions, external contracts need to be called to complete certain operations. The external contract may be a contract that is determined when the caller contract is deployed, or it may be a contract that changes dynamically when the function is executed. Therefore, there is uncertainty in the call sequence between contracts, and the callee contract also has uncertainty.

The reason why re-entrancy attacks can occur is that the function calls the contract deployed by the attacker during execution and runs the attack code to re-entrancy call the contract specified by the attacker. Usually, in order to complete a service, the call relationship between contracts is clear, but in some scenarios, contracts with unknown security will also interact with the caller contract to complete the operation. We will give two examples to illustrate that the feature of calling each other between contracts also provides attackers with the possibility of re-entrancy attacks.

ETH transfer. When ETH transfer is involved between contracts, the sender needs to use *receiver.call.value(amount)("")* to send ETH to the receiver, and the receiver sets the *fallback()* inside its own contract to receive ETH. The *fallback()* [15] is a special function in Solidity. It is a function with no function name, no parameters, and no return value. When a call does not have any data or the called function is a function that does not exist in the callee contract, *fallback()* will be executed automatically. When sending ETH between contracts, if calling with no call data, the caller will invoke callee's *fallback()* to accept ETH. There is no strict standard for the content of *fallback()*, so the attacker can write the re-entrancy attack code into *fallback()*, and when the control flow returns to the attacking contract and invokes *fallback()*, the attacking code will be executed to call other contracts, to steal funds from these contract.

Malicious token transfers. In addition to the circulation of ETH, there are also some tokens that can be traded with each other on Ethereum, so there are token contracts to realize token-related operations and store token-related states. If a transaction involves the transfer of tokens, the corresponding token contract will be invoked to complete the relevant operations during the execution of the transaction and record the state changes. If the token contract is constructed by the attacker, and there is a transfer of tokens in this transaction, then when the contract calls the function of the malicious token contract, the attack code will be executed to re-enter the contract specified by the attacker.

3.3 Shortcomings of Run-Time Detection

Some detection tools choose to perform run-time detection on the transaction, the advantage of this method is that it can obtain the call relationship of the contract when the transaction is executed, and directly detect some operations at the bytecode level when the call sequence is known, without listing all possible calling sequences between functions in a contract or between different contracts, reducing the cost of detection. But this runtime detection method also has some shortcomings.

States transfer between contracts. Different contracts each implement different functions and maintain their own contract states, but if they want to call each other and fulfill the user's needs, they need to transfer state data between contracts. For example, in the Rari Capital attack transaction introduced in Sect. 4, when analyzing the transaction in-depth, we found that although the attacker re-entrant called different contracts, the victim contract A still appeared in the re-entrant call sequence and read its state and returned to the victim contract. None of these tools take this situation into account, but only limit the use of state reading and writing in the same contract.

The opcodes lack some smart contract semantics. The smart contract is written in a high-level computer language, compiled into bytecode, and executed by the EVM. Therefore, the bytecode level can provide more detailed information. For example, there are three opcodes in the EVM to reflect the calling relationship between contracts. They are CALL, DELEGATECALL, and STATICCALL. The function of each call-opcode is different, and the calling relationship between contracts is also different. However, when detecting the bytecode level, there will be omissions in the understanding of the contract semantics level. For example, among the several detection tools we introduced, SLOAD and SSTORE are used to determine whether there are read and write operations on the same state in the contract. But if multiple associated states are involved, their association cannot be judged from a single SLOAD or SSTORE.

4 Classification of Re-entrancy Attacks

Since re-entrancy attacks appeared in 2016, they have occurred continuously and their attack behaviors are constantly changing. Attackers need to plan carefully to avoid transaction rollback due to a wrong operation in a certain step. There are two key points to consider in re-entrancy attacks. First, what kind of target does the attacker choose to avoid defense measures when re-entrancy calls, and can make a profit; what unreasonable use or operations have been carried out in the contract state so that the transaction can be executed normally without being rolled back. We classify re-entrancy attacks according to these two points and combine the actual attack transactions to understand different attack behaviors in-depth.

4.1 Classification by Target of Re-entrancy Calls

When the control flow returns to the attacking contract, the attacking code will be executed and the contract will be called again. Although some contracts have taken re-entrancy attack defense measures, there may still be vulnerabilities exploited by attackers. We discuss the callee contracts and functions that may be re-entered by an attacker into three situations.

Type I. Single-Function Re-entrancy

The earliest incident of theDAO attack was this type. The attacker only targeted the same function in the same contract every time the re-entrant call was made, and this function was related to the user states in the contract. Through an in-depth analysis of theDAO re-entrancy attack transactions [16], we introduce the characteristics of this type of attack behavior.

TheDAO was a decentralized autonomous organization of crowdfunding campaigns, it used to be one of the most popular projects. However, on June 17, 2016, theDAO suffered a re-entrancy attack, losing 3.6 million Ether, which eventually led to a hard fork of the Ethereum network. This is the first re-entrancy attack incident in Ethereum, and there have been more re-entrancy attacks and studies on re-entrancy attacks since then.

Investors deposit their ETH into theDAO to obtain tokens, which they can use to vote on proposals they are interested in, and investors make profits through investment proposals. However, if the user disagrees with the existing proposal, he can split a new child DAO by calling splitDAO() in the DAO contract [17], and the DAO contract will calculate the rewards that the user can obtain, and transfer them to the new child DAO together with the ETH he initially deposited in the child DAO contract. The re-entrancy attack occurs when ETH is transferred.

Figure 1 shows the process of this attack. (1) ***The transaction starts***. This transaction was invoked by an external attacker, calling the pre-deployed attacking contract to carry out the attack. (2) ***Call splitDAO()***. The attacking contract first calls a normal

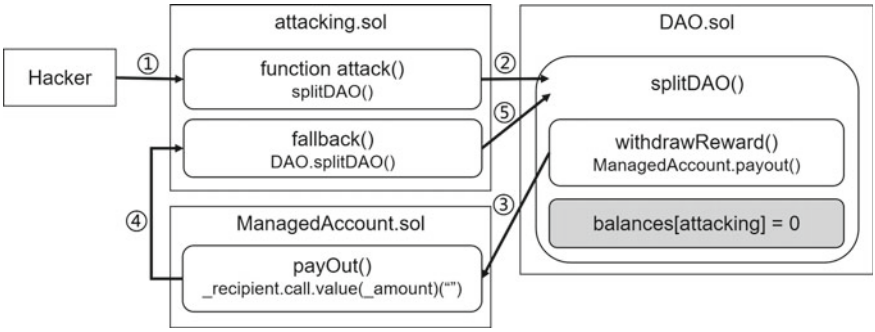

Fig. 1 TheDAO re-entrancy attack

call to the DAO contract. (3), (4) ***ETH transfer***. Before the DAO contract updates the balance states of the attacking contract in this contract, splitDAO() must first send funds to the attacking contract through payOut(). (5) ***Re-entrancy attack***. The attacker writes the attacking code into fallback(), so when the call returns to the attacking contract, the malicious code in fallback() calls splitDAO() in the DAO contract again. Since the balance state of the attack contract has not been modified in (2), that is, although the DAO contract has sent ETH to the attack contract, the DAO contract still stores the old balance state of the attacking contract, so the re-entry call can pass the transfer condition again. Repeating the above operations, the attacking contract gets a large amount of ETH that does not belong to itself from the DAO contract.

From this incident, we can see that the Single-function Re-entrancy Attack is aimed at the same function in the same contract. This function includes user state reading, condition judgment, external calls, and user state modification so that the attacker can change the execution order of the contract through re-entrant calls to complete the theft of funds.

Type II. Cross-Function Re-entrancy

Different from Type I, in this type, the attacker re-entrant calls different functions in the same contract, and these functions operate on the same state. Attackers can use re-entrant calls to disrupt the state modification operations of multiple functions, thereby stealing funds that do not belong to them.

We introduce the behavioral characteristics of Cross-function Re-Entrancy Attacks through an in-depth analysis of Lendf.me re-entrancy attack transactions. On April 19, 2020, a Re-Entrancy Attack occurred in Lendf.me, and the total loss of this attack was about $25 million.

Lendf.me is an instant lending platform. All accounts of Lendf.me are managed by the MoneyMarket contract [18]. Users can deposit assets into the platform through supply() in the contract, and withdraw their own deposits through withdraw(). In supply(), the contract first calculates the changes in states after the user deposits money, and stores the result in a temporary variable. After transferring the user's

assets to the MoneyMarket contract, it will use the calculated temporary variable content to modify the user state. When assets are transferred, the corresponding token contract will be called to record the changes of the user's assets.

Figure 2 shows the process of this attack. (1) ***The transaction starts***. The attacker calls the deployed attacking contract. (2) ***Calls supply() to deposit ImToken tokens***. The ImToken token protocol is compatible with the ERC777 standard. When the token is transferred, tokensToSend() and tokensReceived() in the spender and receiver contracts will be called respectively, so as to complete the token transfer, the signature of both parties, and the notification of the completion of the transaction in one transaction. (3), (4) ***Transfer ImToken token***. The attacking contract is a token spender, and the ImToken contract calls tokensToSend() in the attacking contract. Since there is no specific implementation requirement for tokensToSend(), the attacker can construct malicious tokensToSend(). (5) ***Re-entrancy attack***. We cannot obtain the specific content of the attacking contract, but through in-depth analysis of the transaction [19] execution process, we can find that the attacking contract re-enters the withdraw() in the MoneyMarket contract when executing tokensToSend(). This function will withdraw the previous deposit of the attacking contract in the MoneyMarket contract.

After the re-entry call is completed, it will return to (2) to continue to execute supply(), but because the change value of the user state has been calculated in supply() before the token transfer, and the result stored in a temporary variable, after the re-entrancy attack, the function did not recalculate, but directly updated the user state with the content of the temporary variable, ignoring the withdraw() that occurred in the re-entrant call.

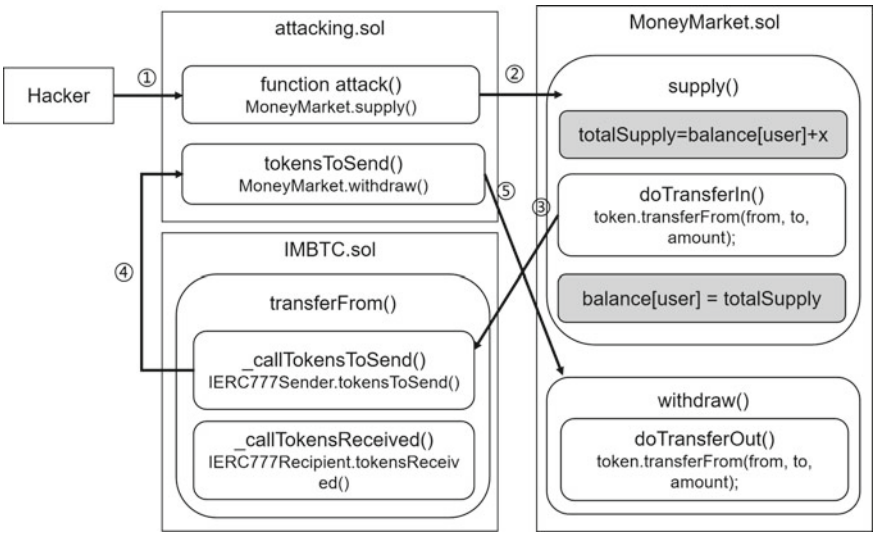

Fig. 2 Lendf.me re-entrancy attack

Type III. Cross-Contract Re-entrancy

In this type of re-entrancy attack, the contract that the attacker re-entrant calls is different from the contract that was called for the first time. There may be a certain relationship between the states of these two contracts. The attacker changes the effect of the state on the function execution through the re-entrant call. The Rari Capital re-entrancy attack transaction that occurred on April 30, 2022, was a Cross-Contract Re-Entrancy attack. The following is our specific analysis of this attack.

Rari Capital is a lending project, users can mortgage the assets they own to obtain the right to borrow. In this project, the Comptroller contract [20] manages the user's information in the lending market. Users can mortgage or redeem assets from the market by calling enterMarket() or exitMarket(). Each asset contract implements operations such as depositing and lending assets.

Figure 3 shows the process of this attack. (1) ***The transaction starts***. The attacker calls the deployed attacking contract. (2) ***Calling borrow()***. The attacking contract calls borrow() in the cEtherDelegate contract to lend ETH. (3) ***Send ETH to the attacking contract***. The cEtherDelegate contract uses to.call.value(amount)("") to send ETH to the attacking contract, calls fallback() in the attacking contract, and executes the attacking code in fallback(). (4) ***Re-entrancy attack***. When executing the fallback(), the attacking contract did not re-enter the function in the cEtherDelegate contract but called exitMarket() in the Comptroller contract to exit the market and redeemed the mortgaged assets. exitMarket() counted the total amount of loans of the attacking contract before redeeming the assets. Due to the fallback() call, the borrow() called in (2) had not yet written the loaned amount of the attacking contract into its borrowing state, so exitMarket() calculates that the loan of the attacking contract is 0, and finally the attacking contract successfully redeemed its mortgage assets and obtained additional loan assets.

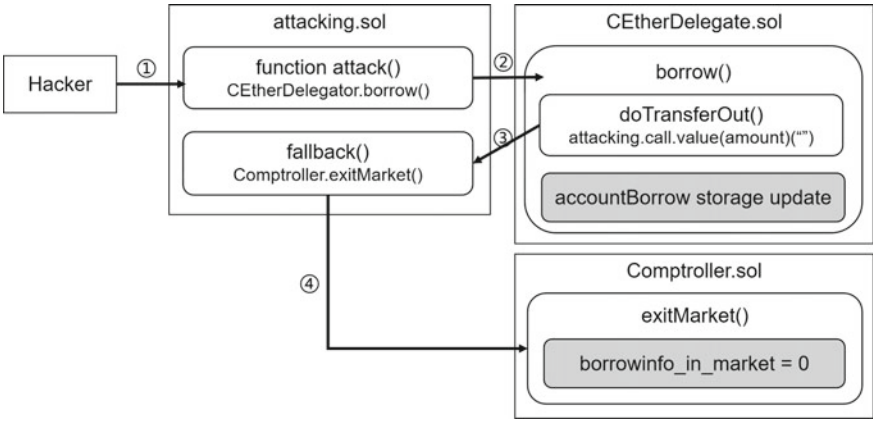

Fig. 3 Rari Capital re-entrancy attack

In this transaction, the same contract was not repeatedly called when the attacking contract was called again. However, after an in-depth analysis of this transaction [21], we found that after the re-entry call to the Comptroller contract, the cEtherDelegate contract appeared in the call stack as a sub-call branch of the Comptroller contract, returned the loan state of the attacking contract and used it for the Comptroller contract to count the total loan amount of the attacking contract.

4.2 Shortcomings of Run-Time Detection

The function of the contract will modify the state stored in the storage during execution, and will also read the data in the storage to complete important operations. The states of user assets are often recorded in the storage. If the attacker only starts an attack transaction to read the state, there is no benefit to be obtained. Therefore, the modification operation of the state is also a key operation used by the attacker during the attack.

Type I. Updating State After Re-entrancy

In this type, the external call occurs before the operation of the victim contract recording the state in the contract storage. Therefore, when the victim contract of the re-entrant call executes the function, the state read from the storage is still not updated.

This is the case for all three attack transactions we introduced in Sect. 4.1. For example, in theDAO attack transaction in Fig. 1, under normal operation, after the DAO contract sends the user's assets to the designated contract, it will immediately return to this contract to record the user's new asset states in storage for subsequent function execution used when. However, after the attacking contract received the assets, it re-entered and called the DAO contract, which interrupted the normal state modification operation, causing the DAO contract to read the old attacker's asset state in the storage when executing the function again, and then sent ETH to the attack contract. The DAO contract does not record the attacker's asset states in the storage until the re-entry call ends, at which point the attacker has successfully received multiple transfers.

Type II. Without State Update After Re-entrancy

Different from Type I, this type does not modify the state after the re-entrant call returns, that is, the attacker does not use the un-updated state in the storage during the re-entrant call, but modifies the state during the re-entrant call. After the call returns, the updated state will be read, but this state will affect the execution result of the original contract, making the result inconsistent with expectations.

We introduce this type of re-entrancy attack through an in-depth analysis of Akropolis re-entrancy attack transactions [22]. The Akropolis reentrancy attack occurred on November 13, 2020, with a loss of approximately $2 million.

Akropolis is an investment aggregator. Users invest their funds and invested protocols in Akropolis, and the platform will automatically choose the path with the highest income for investment to earn the maximum profit for users.

Users can enter the selected protocol, deposited tokens, and their amount into the AdminUpgradeabilityProxy contract [16] through deposit(), and deposit() will calculate the balance of all assets in this contract before and after the user deposits tokens and store the result in temporary variable *balanceBefore* and *balanceAfter*, after determining the user's deposit amount through the difference of the total contract balance, the funds are deposited into the contract and deposit certificate tokens are issued to the user, and the user can use these tokens to withdraw his deposit at any time.

Figure 4 shows the process of this attack. (1) ***The transaction starts***. The attacker calls the deployed attacking contract. (2) ***Calls deposit()***. The attacking contract calls deposit() to deposit the malicious token constructed by the attacker. (3) ***Token transfer***. When transferring tokens, it is required to call back to the token contract and execute transferFrom() to record the tokens transfer from the attacking contract to the AdminUpgradeabilityProxy contract. (4), (5) ***Re-entrancy attack***. Since the token contract is constructed by the attacker, when entering the malicious token contract, the attacking code constructed by the attacker is executed, and the deposit() in the AdminUpgradeabilityProxy contract is called again. This call deposits legal tokens and fully executes deposit(), changes the total balance of the AdminUpgradeabilityProxy contract, and issues a corresponding amount of deposit certificate tokens to the attacker. Return to (2) after the re-entry call to continue to calculate the *balanceAfter* in the deposit() contract.

Since the total balance of the contract changes during the re-entry call, when using to *balanceAfter-balanceBefore* calculate the user's deposit amount, *balanceBefore* is the balance of the contract before the re-entry call executed, *balanceAfter* is the total balance after successfully depositing legal tokens in the re-entrancy attack. After an in-depth analysis of this transaction, we found that although the attacker did not

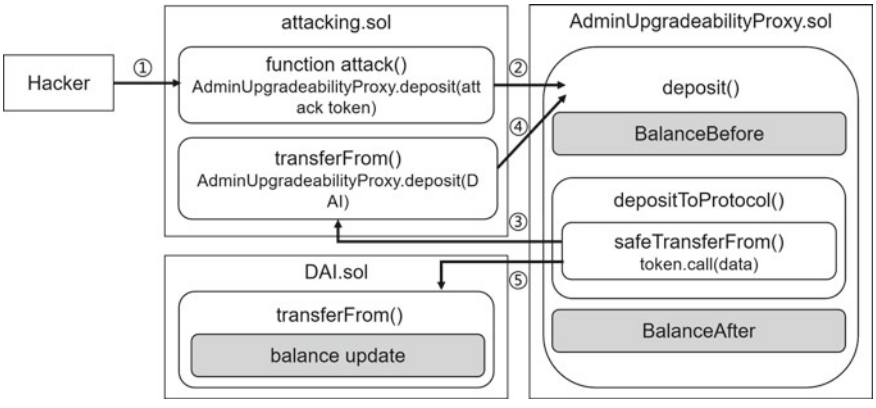

Fig. 4 Akropolis re-entrancy attack

actually deposit assets in the first deposit(), the contract is still based on the result of *balanceAfter-balanceBefore* and issued a deposit certificate token for the attacker. The attacker deposited one asset but got two deposit certificate tokens.

5 Related Work

Starting from the DAO incident, various researches have studied re-entrancy attacks more and more deeply and used many different methods to detect re-entrancy attacks. In this part, we introduce several re-entrancy attack detection tools and analyze their deficiencies or omissions that may occur when detecting attacks.

Akropolis is an investment aggregator. Users invest their funds and invested protocols in Akropolis, and the platform will automatically choose the path with the highest income for investment to earn the maximum profit for users.

In **Sereum** [6], state inconsistency is judged as a re-entrancy attack. The so-called state inconsistency means that the contract already in the call stack is called again. If there is a state that affects the control flow during the execution of this contract, and this state is not modified until it returns to the original contract, it will be considered as an impact. The control flow of the victim contract uses an inconsistent state, so it is determined to be a re-entrancy attack. In the detection, Sereum sets a write-lock for the states that appear in the re-entrant call. If a write operation is performed on a locked state after the call returns, an alarm will be triggered.

SODA [2] believes that if a loop is executed cyclically in a call and ETH is transferred every time the loop is executed, a reentrancy attack occurs in this transaction. A loop means that a call to contract A occurs again in the call sequence initiated by contract A.

Horus [3] also detects circular calls. It thinks that two contract calls in a call stack have the same transaction hash, caller, callee, etc., and they are at different depths of the call stack. In addition, if the contract called for the first time has an operation on SLOAD and SSTORE in the same state during execution, and SLOAD appears before the re-entrant call, and SSTORE appears after the re-entrant call, then it is judged as a re-entrancy attack.

6 Conclusion

In this paper, we discuss re-entrancy attacks from the three aspects of occurrence principle, type, and detection method. Through in-depth analysis, we found that due to the immutability of deployed smart contracts and the characteristics that the contracts can call each other cannot be changed, that's why re-entrancy attacks that cannot be detected by detection tools continue to appear. Therefore, in order to improve the security of smart contracts and blockchains, developers should carefully

use statements or logic that are prone to problems, and use defensive measures appropriately; project parties need to do off-chain audits before contracts are deployed; attack detectors are supposed to conduct comprehensive research on attack behavior before attack detection.

Funding Statement This work was supported by Guangzhou Higher Education Innovation Group 202032854, Guangzhou Science and Technology Plan Project (No. 202102010445).

Conflicts of Interest The authors declare that they have no conflicts of interest to report regarding the present study.

References

1. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: a scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)
2. Chen, T., Cao, R., Li, T., Luo, X., Gu, G., Zhang, Y., Liao, Z., Zhu, H., Chen, G., He, Z., et al.: Soda: a generic online detection framework for smart contracts. In: NDSS (2020)
3. Torres, C.F., Iannillo, A.K., Gervais, A., State, R.: The eye of horus: spotting and analyzing attacks on ethereum smart contracts. In: Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, 1–5 Mar 2021. Revised selected papers, Part I 25, pp. 33–52. Springer (2021)
4. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269 (2016)
5. Qiu, J., Tian, Z., Du, C., Zuo, Q., Su, S., Fang, B.: A survey on access control in the age of internet of things. IEEE Internet Things J. **7**(6), 4682–4696 (2020)
6. Rodler, M., Li, W., Karame, G.O., Davi, L.: Sereum: protecting existing smart contracts against re-entrancy attacks. arXiv preprint arXiv:1812.05934 (2019)
7. Tian, Z., Li, M., Qiu, M., Sun, Y., Su, S.: Block-DEF: a secure digital evidence framework using blockchain. Inf. Sci. **491**, 151–165 (2019)
8. Tian, Z., Luo, C., Qiu, J., Du, X., Guizani, M.: A distributed deep learning system for web attack detection on edge devices. IEEE Trans. Ind. Inform. **16**(3), 1963–1971 (2019)
9. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Security: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018)
10. Zhang, M., Zhang, X., Zhang, Y., Lin, Z.: TXSPECTOR: uncovering attacks in ethereum from transactions. In: USENIX Security Symposium (2020)
11. Ethereum whitepaper. https://ethereum.org/en/whitepaper/ (2023)
12. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. https://github.com/ethereum/yellowpaper. (2022)
13. Evm opcodes. https://www.evm.codes/
14. storage document. https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html (2022)
15. fallback function. https://docs.soliditylang.org/en/v0.8.19/contracts.html#fallback-function (2021)
16. Etherscan: Adminupgradeabilityproxy.sol. https://etherscan.io/address/0x73fc3038b4cd8ffd07482b92a52ea806505e5748
17. Github: Dao.sol. https://github.com/TheDAO/DAO-1.0/blob/master/DAO.sol

18. Github: Moneymarket.sol. https://github.com/Lendfme/contracts/blob/master/contracts/MoneyMarket.sol
19. Tenderly: Lendf.me attack. https://dashboard.tenderly.co/tx/mainnet/0x37085f336b5d3e588e37674544678f8cb0fc092a6de5d83bd647e20e5232897b (2020)
20. Github: Comptroller.sol. https://github.com/Rari-Capital/compound-protocol/blob/master/contracts/Comptroller.sol
21. Tenderly: Rari capital attack. https://dashboard.tenderly.co/tx/mainnet/0xadbe5cf9269a001d50990d0c29075b402bcc3a0b0f3258821881621b787b35c6 (2022)
22. Tenderly: Akropolis attack. https://dashboard.tenderly.co/tx/mainnet/0xe1f375a47172b5612d96496a4599247049f07c9a7d518929fbe296b0c281e04d (2020)