Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.034116

ARTICLE

A Data Consistency Insurance Method for Smart Contract

Jing Deng[1], Xiaofei Xing[1], Guoqiang Deng[2,*], Ning Hu[3], Shen Su[3], Le Wang[3] and Md Zakirul Alam Bhuiyan[4]

[1]School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou, 510006, China

[2]Information Network Engineering and Reasearch Center, South China University of Technology, Guangzhou, 510640, China

[3]Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, 510006, China

[4]Department of Computer and Information Sciences, Fordham University, New York, 10458, USA

*Corresponding Author: Guoqiang Deng. Email: denggq@scut.edu.cn

Received: 06 July 2022 Accepted: 09 November 2022 Published: 08 October 2023

ABSTRACT

As one of the major threats to the current DeFi (Decentralized Finance) ecosystem, reentrant attack induces data inconsistency of the victim smart contract, enabling attackers to steal on-chain assets from DeFi projects, which could terribly do harm to the confidence of the blockchain investors. However, protecting DeFi projects from the reentrant attack is very difficult, since generating a call loop within the highly automatic DeFi ecosystem could be very practicable. Existing researchers mainly focus on the detection of the reentrant vulnerabilities in the code testing, and no method could promise the non-existent of reentrant vulnerabilities. In this paper, we introduce the database lock mechanism to isolate the correlated smart contract states from other operations in the same contract, so that we can prevent the attackers from abusing the inconsistent smart contract state. Compared to the existing resolutions of front-running, code audit, and modifier, our method guarantees protection results with better flexibility. And we further evaluate our method on a number of de facto reentrant attacks observed from Etherscan. The results prove that our method could efficiently prevent the reentrant attack with less running cost.

KEYWORDS

Blockchain; smart contract; data consistency; reentrancy attack

1 Introduction

Blockchain technology integrates encryption algorithms, distributed ledger technology, and other technologies to establish and maintain a decentralized and trust-free database. It has the characteristics of information transparency and tamper-proof, which makes it have broad prospects of application [1]. As one of the most popular blockchain platforms, Ethereum [2] introduced the concept of smart contracts. A smart contract is a program that runs on the blockchain and has the functions of storing data and executing transactions. It can also be regarded as a distributed ledger with monetary attributes [3]. It may be applied to a variety of aspects, including token assets, payments, supply chains, and electronic bills. In Ethereum, 10.7 million smart contracts were created and 345 million transactions were made in 2020 [4]. Since smart contracts have financial properties and can hold a significant sum of money, they are more likely to attract the attention of malicious people. Besides, due to the immutability of the blockchain, vulnerabilities in smart contracts are difficult to repair after being discovered and attacked, resulting in more serious losses.

In the smart contract, because the developer's design is not rigorous enough, the attacker may be able to exploit reentrancy and other methods to generate data inconsistency, launching an attack and incurring significant financial losses. The infamous the DAO (Decentralized Autonomous Organization) attack [5] exploited reentrance flaws to inflict nearly $60 million in losses and even led to the fork of the Ethereum blockchain, which severely depreciated the digital currency and had a catastrophic impact on the market. In addition, Cream.Fiance suffered an attack due to a reentrance vulnerability in August 2021, resulting in a loss of approximately $18.8 million.

The defense methods for this vulnerability are divided into the following types: 1) Identify vulnerabilities in smart contracts using analysis tools and use some methods to fix them. For example, the study by Qian et al. [6], the study by Liu et al. [7], and the study by Samreen et al. [8]. 2) Intercept harmful transactions by detecting malicious behavior based on runtime information, such as ÆGIS [9] and a defense technology proposed by Xiang et al. [3]. 3) Reinforce the security of smart contracts in the stage of code writing, such as the ReentrancyGuard modifier of Openzeppelin [10] and a security function based on the state lock in a patent called "A design method and system for security function of smart contract based on state lock". Among the first two types of methods, some studies intercept the attack through EVM. But this way of defending against attacks has to do with the blockchain nodes, and changing the clients of each node is something that has a significant impact on the blockchain [11,12]. Some methods are to inspect the contract and identify if there is a vulnerability or an attack. It does not identify all reentry attacks and may not be defensible if a new type of reentry attack is encountered.

The method in this paper belongs to the third category, which is to reinforce the security of smart contracts during the stage of code writing, such as ReentrancyGuard. When a reentry attack occurs, it relies on the contract itself to restrict the attacker's behavior and prevents the attack from being successfully executed. This method is convenient and straightforward to use, but the current studies can only partially avoid the reentrancy problems, and there is still a lack of exploration in this area. Therefore, we want to find a better method that can defend against more types of reentrancy attacks. This method has to be practical and low-cost.

Our contribution in this paper includes two folds. First, we introduce the idea of modifying the correlating smart contract states as a unit to prevent the abuse of dirty states by reentrant attacks. Second, we propose to utilize external smart contracts to record the editing circumstances of the correlating smart contract states, which would not be affected by reentrant attacks. Finally, we prove the feasibility of our method by deploying our method on the smart contract which has already suffered from reentrant attacks, and our experiments indicate that our method could effectively ensure data consistency with better efficiency.

2 Preliminary

2.1 Smart Contract

There are two types of accounts in Ethereum, external accounts and contract accounts. External accounts are accounts owned by users. Contract accounts are accounts allocated for smart contracts and controlled by the code of the contract. Technically, a smart contract can be seen as a set of functions defined by bytecode instructions that can receive, store, and send messages and currencies. Developers can use high-level programming languages (e.g., solidity) to write smart contracts. Solidity is currently the most commonly used language for smart contract development, and it is a Turing-complete high-level programming language. The work in this paper uses Solidity language to realize the writing of the contract. The code is compiled to get bytecode, which is then deployed to the blockchain. After the contract is deployed on the chain, the publisher cannot modify the contract and anyone can interact with the contract. There are variables stored in the contract and state variables are variables whose values are permanently stored in contract storage.

External accounts can initiate transactions and call contracts, and contract accounts can call each other as well. The state of the accounts can be changed through transactions. After the transaction is created, it will be broadcast to other Ethereum nodes, executed by miners, and packaged into blocks [13,14]. In order to prevent malicious transactions and regulate miners' behavior, each operation in Ethereum charges a certain fee in units of gas.

2.2 Reentrancy Attack

When a user calls a smart contract, there may be some operations that the contract calls an unknown contract given by the user. The external calls are easily exploited by attackers. The attacker can construct a malicious contract so that the object of the external call is the function in the malicious contract. Then the control is transferred to the malicious contract. In the called function of the malicious contract, the attacker can call the contract of the attacked project once more, and launch attacks like multiple transfers until the gas or contract fund runs out. There are many ways to launch reentrancy attacks. According to the research of Rodler et al. [15], there are several types of reentrancy attacks, including same-function reentrancy, cross-function reentrancy, delegated reentrancy, and create-based reentrancy. In same-function reentrancy, the victim contract is reentrant in the same function, which is the most classic reentrancy attack. In cross-function reentrancy, the victim contract is reentrant in a different function. In delegated reentrancy, operations that can be reentrant (such as transfers via call()) exist in the function called via the delegate call. In create-based reentrancy, the victim contract creates a contract, and the constructor of the new contract will call the function in the malicious contract.

2.3 Data Inconsistency

In a database, it is important to ensure data consistency. Due to improper concurrency control, data inconsistency may occur in the database, causing problems like dirty writing and lost updates. In smart contracts, the problem of data inconsistency also exists.

In smart contracts, we refer to the logical association constraints between the state variables of the smart contract that particularly undertakes the application business as data consistency. For example, the constraint between "account" and "balance" is the number of tokens held by each project participant. When smart contracts perform data operations, especially complex data operations, an intermediate state that does not meet the associated constraints will appear. And this is the state of data inconsistency.

When a smart contract is called externally in a dirty state, the external contract has the opportunity to maliciously cause data inconsistency of the state variable through re-entry, or use the original data inconsistency state to break the original constraints of the smart contract, tamper with the control flow and state of the original smart contract. This data inconsistency problem is usually exploited by attackers to attack the contract itself through the fallback function or other ways to achieve malicious attacks like repeated transfers. In a reentrancy attack, the successful execution of the reentrancy operation requires that the original constraints in the smart contract are satisfied. So in the reentrant attack, one or more relevant state variables involved in this constraint condition are in a data-inconsistent state. Therefore, we believe that solving the data consistency problem in smart contracts can effectively prevent reentrancy attacks.

The data consistency mentioned here is in line with the data consistency of traditional relational databases. In the relational database, the goal of ensuring data consistency is refined into ACID principles, namely: 1) Atomicity. The transaction is either all committed successfully or all failed to roll back. There is no partial success state and cannot perform only part of the operation. 2) Consistency. The execution of the transaction cannot destroy the integrity and consistency of the database data. The database must be in a consistent state before and after a transaction is executed. 3) Isolation. Concurrent transactions are isolated from each other, and the execution of a transaction cannot be interfered with by other transactions. 4) Durability. Once a transaction is committed, the corresponding data state changes will be permanently stored.

In smart contracts, because blockchain data is difficult to tamper with, it is clear that the principle of durability can be satisfied. Since the smart contract transactions are executed serially and the transaction itself is executed atomically, a qualified developer can also basically guarantee the other principles if no reentry attacks are considered so that the data of the transaction results under normal circumstances are in a consistent state and satisfy the principle of consistency. Therefore, the key to guaranteeing data consistency in smart contracts is to ensure that sequences of operations (including reading and writing operations, called correlated sequences of operations) on the same set of logically related states are executed atomically and isolated from each other. Thus, it satisfies the principles of atomicity and isolation.

3 Methodology

In this paper, we use mutual exclusion locks to isolate data read/written between a set of associated operations. As shown in Fig. 1, before performing a set of associated operations, it is necessary to apply for locking the state variables accessed by the operations, and the operations can be performed only after the locking is successful. The corresponding state variables cannot be accessed again until they are unlocked. In this way, a state variable cannot be accessed by more than one application in a single association operation, preventing the state variable from being accessed in inconsistent states.

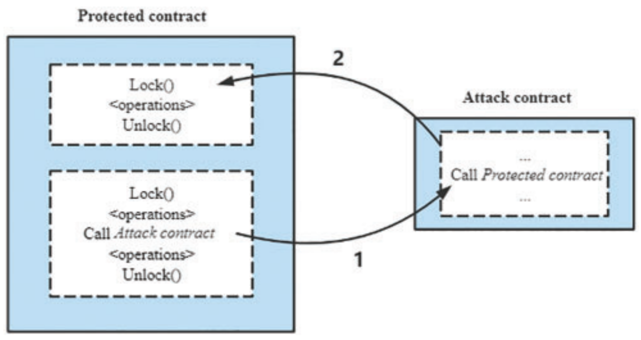


Figure 1: Data consistency guarantee mechanism

After the operation sequence is finished, the state variables should be unlocked, and the locked state variables should be released so that they can be requested in the next operation sequence. If a state variable is locked and an external contract attempts to access the data, the application fails and no subsequent operations are performed. To achieve atomic execution of the associated operation sequence, i.e., no partially successful state exists, the unreleased locked state variables are checked before the smart contract returns. If an unreleased state variable exists, the transaction will be rolled back. In this way, access to resources can be turned into a mutually exclusive operation through the lock, and the consistency of state data in smart contracts can be guaranteed.

In this paper, we implement mutual exclusion locks in a smart contract called "Locker" and allocate mutually exclusive resources based on smart contracts' addresses and names of state variables. Adding and releasing a mutual exclusion lock is done by calling the method in the Locker contract. In the method implementation described in this paper, there is the lock function and the unlock function.

First, we need to clarify when and how the functions will be used. To ensure data consistency, it is necessary to prevent state variables from being read or written in dirty states in a sequence of operations on logically related variables. Thus, the lock function is called before operating on state variables (i.e., variables that will probably be in a dirty state). Meanwhile, since the read state variable may be used directly or indirectly as judgment conditions, and may also be used for data updates or other purposes, it is necessary to detect the lock state of the variables before reading them. This is done to prevent smart contracts from performing operations that deviate from the contract's original logic because of the incorrect state variables.

The lock function is called before accessing state variables, and a lock can be applied to one or more variables at a time. In the lock function, the lock state of the state variables is detected before the locking operation is performed, i.e., to detect whether the requested state variables have been locked. If none of the state variables have been locked, the lock function records the data of this visit (the recorded data is called "access record") and returns true. Then the group of state variables can be accessed. If there are one or more state variables that have already been locked, return false. The above operations prevent data from being accessed repeatedly and achieve the purpose of locking data.

The above detection is implemented in combination with the reentry depth. When a locking operation is performed on a set of state variables, the access data of the external contract is detected and recorded, and the access records can be used in subsequent detection to determine whether the external contract reenters in a certain sequence of associated operations in that transaction. In order to accurately record the locking status of variables in different contracts, the state variable's name and its contract address information are included in an access record. The storage and query of access records can be implemented in various ways. In this paper, we implement a Locker contract, in which the lock method and unlock method are implemented.

When the lock function is called, based on the contract address and the state variable name, it detects whether the requested variables have already been recorded. If no record exists, the lock request is successful. The corresponding access record is recorded in the Locker contract (that is, the corresponding state variables become locked), and true is returned. Otherwise, when one or more state variables in the set of requested variables are in the locked state, the application for locking fails, and false is returned. When locking fails, we assume that a reentrant operation occurred in the transaction, so we can avoid the potential reentrant attack by means such as rolling back the transaction. After the operation on an associated set of state variables is finished, the unlock function is called to delete the corresponding access record. Then, this set of state variables can be requested to be accessed (back to the unlocked state). The flow chart is shown in Fig. 2.

As the example in Fig. 3, there is a Loaner contract, where the totalDebt and loanLimit are state variables in it. External contracts can deposit a guarantee to the Loaner contract and obtain the loan limit by calling depositGuarantee(). When loan() is called, the Loaner contract transfers tokens to the caller. And when the totalDebt is 0, the caller can take out the guarantee by calling withdrawGuarantee().

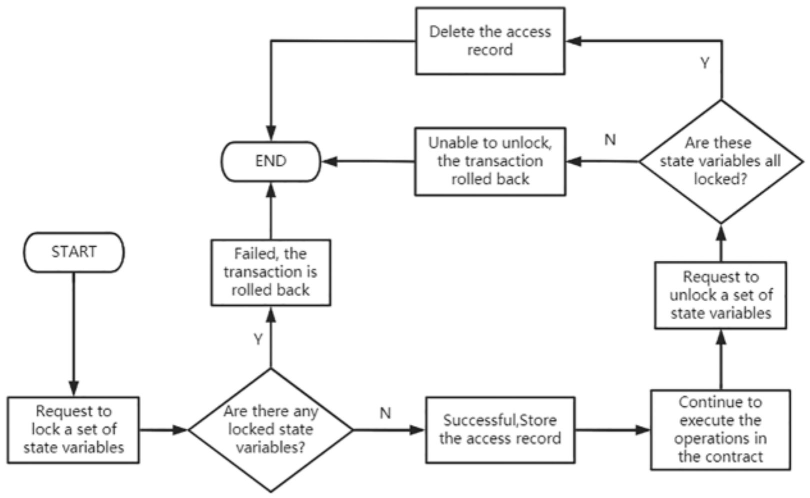
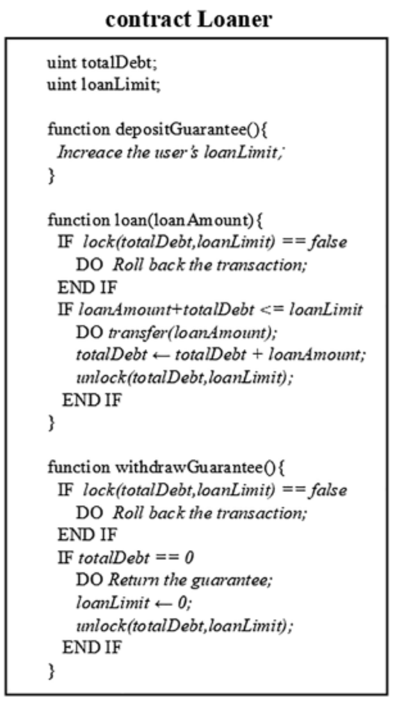

Delete the access record
Y
END
Unable to unlock, the transaction rolled back
N
Are these state variables all locked?
START
Failed, the transaction is rolled back
Request to unlock a set of state variables
Y
Request to lock a set of state variables
Are there any locked state variables?
N
Successful,Store the access record
Continue to execute the operations in the contract

Figure 2: Flow chart of locking and unlocking state variables



```
contract Loaner

uint totalDebt;
uint loanLimit;

function depositGuarantee(){
  Increace the user's loanLimit;
}

function loan(loanAmount){
  IF lock(totalDebt,loanLimit) == false
     DO Roll back the transaction;
  END IF
  IF loanAmount+totalDebt <= loanLimit
     DO transfer(loanAmount);
     totalDebt ← totalDebt + loanAmount;
     unlock(totalDebt,loanLimit);
   END IF
}

function withdrawGuarantee(){
  IF lock(totalDebt,loanLimit) == false
     DO Roll back the transaction;
  END IF
  IF totalDebt == 0
     DO Return the guarantee;
     loanLimit ← 0;
     unlock(totalDebt,loanLimit);
   END IF
}
```

Figure 3: Contract loaner

When a reentrant attack occurs, the attack contract calls loan() in the Loaner contract after putting in the guarantee, and the fallback function of the attack contract is called after the transfer occurs and before the totalDebt is updated, at which time the attack contract can take over the control flow and then call withdrawGuarantee() function. In this case, if the lock function is not used, totalDebt and loanLimit can be reentered and the attack contract can successfully call the withdrawGuarantee(), causing a loss to the Loaner contract. If the lock function is used, we need to apply a lock for totalDebt and loanLimit first when calling the withdrawGuarantee(). Since totalDebt and loanLimit are already locked, false is returned and reentrant attacks are avoided.

4 Evaluation

In this section, we design experiments to test the effectiveness and performance of the method proposed in this paper. All the experiments are conducted on a computer equipped with an AMD Ryzen 7 4800H CPU at 2.90 GHz and 32 GB RAM. The language used in the experiments is Solidity. We use Remix to compile the contract. After debugging the contract, we can publish the contract to any blockchain that supports Solidity smart contracts. In the experiment, we chose to test on the Ethereum test network Rinkeby.

In order to verify the types of reentrancy attacks that this method can defend against, we refer to the types of reentrancy attacks in the research of Rodler et al. [15] and the experimental contract published on github by the research of Torres et al. [9]. Cross-function reentrancy, delegated reentrancy, and create-based reentrancy attacks are three forms of reentrancy attacks used to conduct experiments. In addition, we selected The DAO and SpankChain from publicly reported attacks and found their source code from Etherscan. We wrote the contract SimpleDAO based on the attack logic in The DAO, then deployed SimpleDAO contract and LedgerChannel contract in the SpankChain event to the test net. Next, we edit the two contracts, add the lock and unlock functions before and after some operations, and deploy the edited contract to the test net. We write attack contracts based on the actual reports of how hackers attacked these contracts, then attack unlocked contracts and locked contracts respectively. After testing, it is found that the contract using the lock function and unlock function can successfully avoid the attack. It verifies that our method can defend against the above three types of reentrancy attacks, and shows that our method is also effective in selected real reentrancy attacks that have actually occurred. The test information such as the increased line of code (LOC) for these reentry attacks is shown in Table 1 below.

Table 1: Test information for three types of reentry attacks

Event	Test contract	Compiler version	Increased LOC
Cross-function reentrancy	CrossFunctionReentrancy	0.5.0	18
Delegated reentrancy	Bank	0.5.0	14
Create-based reentrancy	Bank	0.5.0	15
The DAO	SimpleDAO	0.4.19	8
SpankChain	LedgerChannel	0.4.23	16
Uniswap	UniswapV2Router02	0.6.6	16

We also found the source code for Uniswap from Etherscan and added codes to the functions of adding liquidity and swapping tokens, to lock and unlock the state variables involved in these functions. During this process, we performed 6 experiments for each function, recording and comparing the gas consumption and increased LOC before and after adding codes. The gas consumption is shown in Fig. 4. In these transactions, the average increase in gas consumption is 15%.

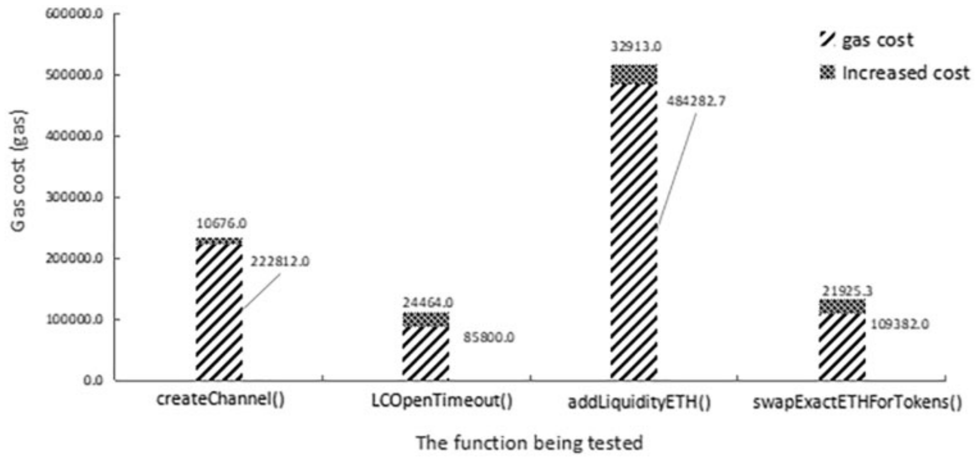


Figure 4: Gas cost of the method

The cost in our method originates mainly from calling functions and storing data, and it is related to the number of variables to be locked and the number of locks to be added. When the number of locks and the number of locked variables do not vary much, we believe that the original business complexity of the function has a significant impact on the percentage increase in gas, and therefore only analyzing the percentage increase in gas is not comprehensive enough. Thus, we selected multiple state variable names from the contracts covered in the experiments above, wrote contracts to perform lock and unlock operations on different numbers of state variables, and use the gasleft() function to test and get the overhead of the lock and unlock functions. The result is shown in Fig. 5, when more variables are locked at once, the average gas consumption of locking each variable is significantly lower.

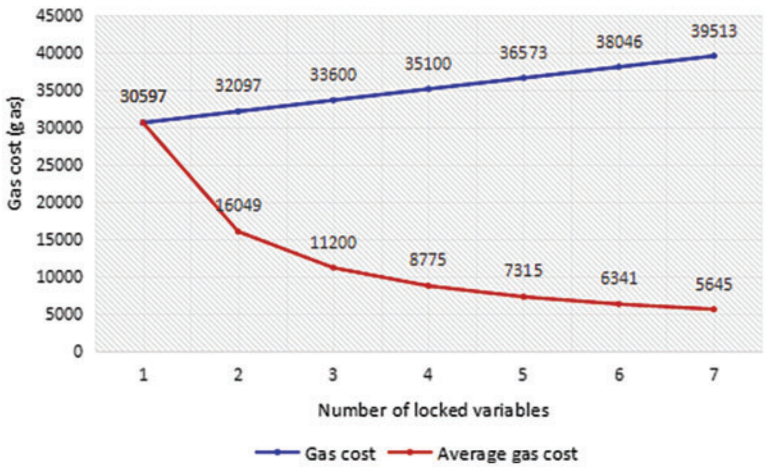


Figure 5: Gas cost with different numbers of locked variables

In addition, we implement an access control contract for the cost comparison with our method. We implemented a contract with access control features based on existing defense methods [3], including transit allowed list, transit banned list, transit restricted list, and interface transit time restriction. We also modified the LedgerChannel contract to use the access control contract and experimented with the cost of running this defense approach in the same environment in normal times. A comparison of the cost of the access control approach and our approach when no attack occurs is shown in Fig. 6. It can be seen that our method consumes less than the access control contract when no attack occurs. And in case of an attack, our approach does not incur more cost consumption. In contrast, in the experiments in the above study, the defensive behavior in case of an attack is to restrict the attacker's address or restrict the interface by initiating a transaction. This generates additional consumption and is more expensive because the transaction is set at twice the gas price of the attack transaction.

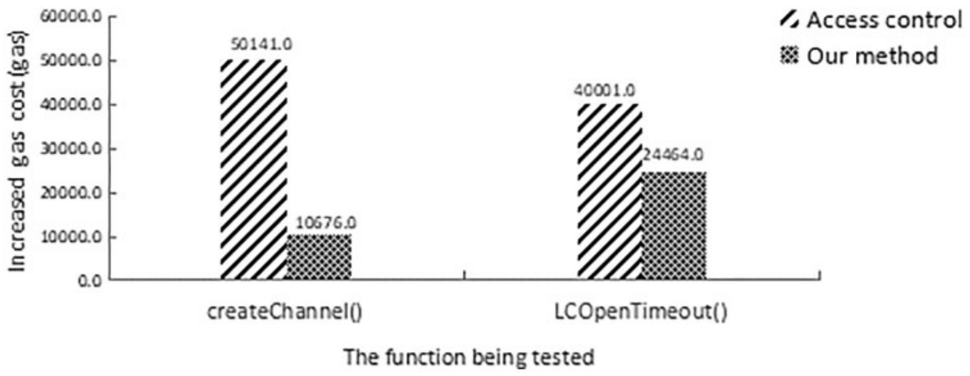


Figure 6: Comparison of increased gas cost

In terms of interception success rate, although the defensive transaction sets a higher gas price in the access control method, there is still a possibility that the defensive transaction is packaged on the chain after the attacking transaction. This is because the defensive transaction is sent after the attacking transaction is detected. On the Rinkeby test chain, the block time is about 15 s. And the average time of the interval between attack and defense transactions in the above research method is 0.533 s. The interval time is longer considering that there is an uncertain transmission time between the initiation of a transaction and its delivery to a node. Therefore, we conduct experiments to continuously obtain the timestamp of the latest block to infer the timestamp of the next block. And we issue transactions at different intervals before the predicted time to see if this transaction can be packaged in the next block.

To avoid the experiment being affected by the order in which transactions are packaged, we set a high gas price. We calculate the probability that a transaction issued at each time point is packed in the next block, as shown in Fig. 7. We use the probability to calculate the weighted average of the transmission time of 1.783 s. This means that there is a time difference of 2.316 s between the detection of an attack and the transmission of a defensive transaction to the node. Therefore, it can be calculated that the probability of two transactions being packed into the same block is 84.6%. Of course, due to network latency, the transmission time may vary significantly from one environment to another, making the interception success rate change as well. We also conducted experiments on the Rinkeby test chain, setting the transmission interval between attacking and defending transactions to 0.533 s. And the two transactions are called a set of transactions. We send a set of transactions 75 times, of which there are 9 defense failures, i.e., the defense success rate in the experiment is 88%. While our method does not need to initiate additional transactions, which are all successfully defended in the experiments of this section.

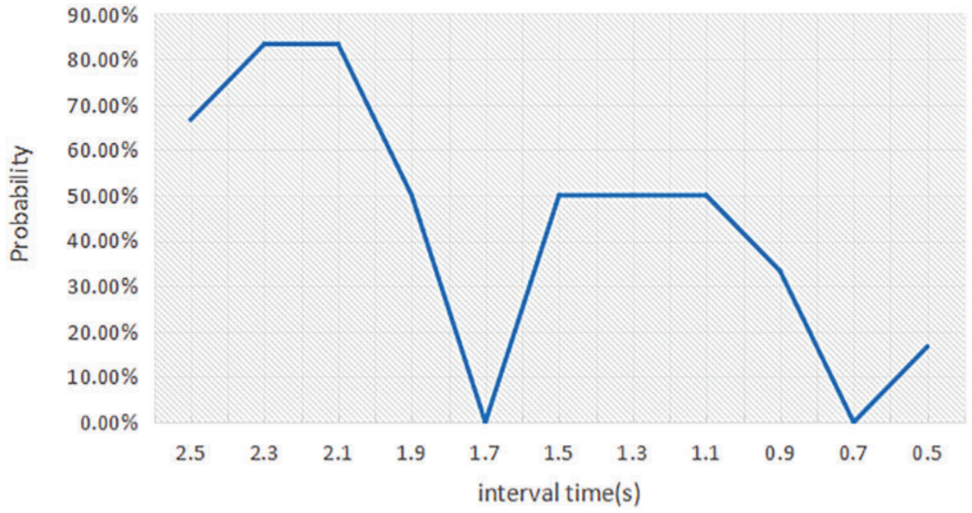


Figure 7: The probability that the transactions be packed in the next block

5 Related Work

This section introduces the defense research on smart contract reentrancy attacks. Since the method in this article belongs to the coding stage, it will be compared with other methods in the coding stage. In the current research on reentrancy attacks, using analysis tools to uncover reentrance vulnerabilities helps with adjustments to the smart contract before being deployed to the chain. There are tools that analyze contracts statically, such as analyzing the contracts written in Solidity or the bytecodes of the contracts. Qian et al. proposed a reentrancy detection method based on a sequential model [6], Liu et al. proposed a fuzzing-based analyzer to automatically detect reentrancy bugs [7], and Samreen et al. proposed a framework that combines static and dynamic analysis to capture reentry attacks [8].

Some tools can detect attacks based on runtime information. Since it tracks the state of the contract in real time, it has more basis for judgment than static analysis of the contract. However, if the real-time data is obtained by staking to the contract code, it will undoubtedly bring higher gas consumption. Some studies such as Sereum are able to detect attacks and intercept malicious transactions in real time by modifying the EVM. However, it would require all miners to bundle the Ethereum clients with the tool, which is difficult to implement.

All of the above methods of detection tools need to analyze the control flow and other characteristics of existing vulnerabilities or attacks, there may be new reentry attacks that are not identified. Some of the methods cannot be updated after the contract uses them and the contract is deployed, and some of the methods that can be updated need to be updated in time before the attack occurs to avoid losses.

The defense methods in the contract code-writing phase also need to be used before the contract is deployed on the chain. Based on the analysis in Section 2.2, we believe that the key to guaranteeing the consistency of smart contract data at this stage is to ensure that the dirty state is not maliciously exploited. We think there are two solutions to guarantee the consistency of smart contract data in the code-writing phase: 1) Ensure that the smart contract does not make (or cannot make) external calls in a dirty state. 2) Ensure that the smart contract cannot access (read or write) the dirty state again when making external calls.

For the first approach, the current main solution is to restrict external calls. Since smart contracts consume a certain amount of gas for each operation, one common way is to use send() or transfer() functions when sending tokens to external addresses. This method limits the number of gas so that the gas is not sufficient to call the contract again. Therefore, this method prevents reentrant attacks by limiting the number of calls. However, this approach is not flexible enough and may also cause problems when calling callback functions due to insufficient gas. Since every opcode supported by Ethereum Virtual Machine (EVM) has a gas cost and it is not constant, this may cause the contract with the gas limit set to not execute smoothly after the gas cost is adjusted.

Checks-Effects-Interactions [16] is also a commonly mentioned way that follows a sequence of checking all preconditions first, then changing the contract state, and interacting with other contracts in the last step. This approach ensures that states change before external calls by strictly checking the interaction and execution order of statements, so that all state changes are completed before any potential reentry point, thus ensuring state consistency. This approach requires a high level of audit correctness and is susceptible to human error.

The ReentrancyGuard approach in the OpenZeppelin smart contract codebase provides a modifier that, when applied to a function, can make the function "non-reentrant" by failing the operation when reentry is detected. However, since the modifier is applied to functions, it guarantees that the same function cannot be called more than once. But there is still a risk that the contract will be reentrant if the same data is accessed through different functions.

In the second type of method, making the dirty state inaccessible to external contracts is achieved through locks. For example, the safe function proposed by Chen et al. Since the function call() has no gas limit setting, it is easy to be exploited to read dirty states for reentry attacks. This method avoids the call function from being exploited by attackers by designing the safe library function SafeCall() so that the state data is locked when the call function is used and the lock is released at the end.

Current research focuses on modifying the EVM or analyzing the contract code. They detect the control flow in the contract and analyze the code or transaction characteristics to determine if there is a reentrant vulnerability. The difference between our method and the above methods is that our method does not directly detect vulnerabilities, but provides a method to be used in smart contracts. It ensures data consistency within the smart contract so that while performing external calls, the smart contract cannot re-access state variables that are in an inconsistent state. Thus, in the event of a re-entry attack, we are able to roll back the transaction and the attack fails. In addition, other methods cannot solve the smart contract data consistency problem effectively. The restriction on external calls reduces the flexibility of development, which in turn limits the application mode of smart contracts. The current state-lock-based method only targets call(), and it cannot be defended if attackers use other functions to attack, such as create-based reentrancy. For the purpose of preventing reentrancy attacks more comprehensively, the data consistency insurance method proposed in this paper ensures that smart contracts cannot access dirty states again in external calls [17]. This method does not result in situations where external calls cannot occur. While improving the flexibility of development, it ensures that operations on the same set of logically related states are executed atomically and isolated from each other, thus guaranteeing the consistency of smart contract data.

6 Conclusion and Future Work

In this paper, we propose a method to ensure the consistency of smart contract data. It is based on the mutual exclusion lock and achieves isolation of data read/write between sequences of associated operations. It detects the lock state combined with the reentry depth to judge whether it is reentrant or not. It ensures that external contracts cannot read or write to the corresponding state variables when a contract performs a data operation with a dirty state. So data consistency before and after the operation sequences is guaranteed. This method also provides a novel idea for the defense of reentrant vulnerabilities.

In addition, the existing limitations of this method need to be pointed out. This method restricts all reentrant operations, even if the operation is not malicious. And there is an additional gas cost when using this method, and the cost increases when there are many operations of locking. But for contacts that may suffer big losses, such attrition is acceptable. Besides, in the experiments of this paper, analyzing contracts and adding locks are implemented manually. The above problems may be improved in subsequent work.

Acknowledgement: The authors are grateful to the financial supports from the National Key Research and Development Plan, Key-Area Reasearch and Development Program of Guangdong Province, CCF-NSFOCUS Kunpeng Scientific Research Fund, National Natural Science Foundation of China, Guangdong Higher Education Innovation Group, and Guangzhou Fundamental Reasearch Plan of "Municipal-School" Jointly Funded Projects.

Funding Statement: This work was supported by National Key Research and Development Plan (Grant No. 2018YFB1800701), Key-Area Research and Development Program of Guangdong Province 2020B0101090003, CCF-NSFOCUS Kunpeng Scientific Research Fund (CCF-NSFOCUS 2021010), National Natural Science Foundation of China (Grant Nos. 61902083, 62172115, 61976064), Guangdong Higher Education Innovation Group 2020KCXTD007 and Guangzhou Higher Education Innovation Group (No. 202032854), and Guangzhou Fundamental Research Plan of "Municipal-school" Jointly Funded Projects (No. 202102010445).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Xiaofei Xing, Guoqiang Deng, Ning Hu, Shen Su; data collection: Jing Deng, Ning Hu; experiment: Jing Deng, Xiaofei Xing, Guoqiang Deng; analysis and interpretation of results: Le Wang and Md Zakirul Alam Bhuiyan; draft manuscript preparation: Jing Deng, Shen Su. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data used in this study will be shared on reasonable request to the corresponding author. However, due to privacy and confidentiality agreementd, certain data may not be released or made publicly available.

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

References

[1] H. Lu, C. Jin, X. Helu, C. Zhu, N. Guizani *et al.,* "AutoD: Intelligent blockchain application unpacking based on JNI layer deception call," *IEEE Network*, vol. 35, no. 2, pp. 215–221, 2020.
[2] V. Buterin, "Ethereum whitepaper," 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] J. Xiang, Z. Yang, S. Zhou and M. Yang, "A runtime information based defense technique for Ethereum smart contract," *Journal of Computer Research and Development*, vol. 58, no. 4, pp. 834, 2021.

[4] H. Kamarul, "Ethereum in 2020: The view from the block explorer," 2021. [Online]. Available: https://medium.com/etherscan-blog/ethereum-in-2020-the-view-from-the-block-explorer-2f9a1db2ee15

[5] V. Buterin, "CRITICAL UPDATE Re: DAO Vulnerability," 2016. [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

[6] P. Qian, Z. Liu, Q. He, R. Zimmermann and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19685–19695, 2020.

[7] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen *et al.,* "ReGuard: Finding reentrancy bugs in smart contracts," in *40th ACM/IEEE Int. Conf. on Software Engineering (ICSE)*, Gothenburg, Sweden, pp. 65–68, 2018.

[8] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in Ethereum smart contracts," in *2020 IEEE Int. Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, London, Canada, pp. 22–29, 2020.

[9] C. F. Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker *et al.,* "Ægis: Shielding vulnerable smart contracts against attacks," in *Proc. of the 15th ACM Asia Conf. on Computer and Communications Security*, Taipei, Taiwan, pp. 584–597, 2020.

[10] F. Giordano and A. Bachfischer, "OpenZeppelin-Contracts," 2021. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts

[11] C. Li, M. Dong, J. Li, G. Xu, X. Chen *et al.,* "Healthchain: Secure EMRs management and trading in distributed healthcare service system," *IEEE Internet of Things Journal*, vol. 8, no. 9, pp. 7192–7202, 2021.

[12] G. Xu, J. Dong, C. Ma, J. Liu and U. G. O. Cliff, "A certificateless signcryption mechanism based on blockchain for edge computing," *IEEE Internet of Things Journal*, 2022.

[13] C. Li, M. Dong, J. Li, G. Xu, X. Chen *et al.,* "Efficient medical Big Data management with keyword-searchable encryption in healthchain," *IEEE Systems Journal*, vol. 16, pp. 5521–5532, 2022.

[14] Z. Tian, M. Li, M. Qiu, Y. Sun and S. Su, "Block-DEF: A secure digital evidence framework using blockchain," *Information Sciences*, vol. 491, pp. 151–165, 2019.

[15] M. Rodler, W. Li, G. O. Karame and L. Davi, "Sereum: Protecting existing smart contracts against Re-entrancy attacks," in *26th Annual Network and Distributed System Security Symp. (NDSS)*, San Diego, pp. 1–15, 2019.

[16] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the Ethereum ecosystem and solidity," in *2018 Int. Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Campobasso, Italy, pp. 2–8, 2018.

[17] C. Li, Y. Tian, X. Chen and J. Li, "An efficient anti-quantum lattice-based blind signature for blockchain-enabled systems," *Information Sciences*, vol. 546, pp. 253–264, 2021.