frontiers
in Computer Science

ORIGINAL RESEARCH
published: 17 February 2021
doi: 10.3389/fcomp.2021.598780

A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks

Ayman Alkhalifah[1]*, *Alex Ng*[2], *Paul A. Watters*[2] *and A. S. M. Kayes*[3]

[1]Ayman Alkhalifah, Riyadh, Saudi Arabia, [2]Department of Security Studies and Criminology, Macquarie University, Sydney, NSW, Australia, [3]Department of Computer Science and Information Technology, La Trobe University, Melbourne, VI, Australia

In Ethereum blockchain, smart contracts are immutable, public, and distributed. However, they are subject to many vulnerabilities stemming from coding errors made by developers. Seven cybersecurity incidents occurred in Ethereum smart contracts between 2016 and 2018, which led to financial losses estimated to be over US$ 289 million. Reentrancy vulnerability was the cause of two of these incidents, and the impacts went far beyond financial loss. Several reentrancy countermeasures are available, which are based on predefined patterns that are used to prevent vulnerability exploitation before the deployment of a smart contract; however, several limitations have been identified in these countermeasures. Motivated by all these issues, the objective of this article is to help developers improve the cybersecurity of smart contracts by proposing a solution that calculates the difference between the contract balance and the total balance of all participants in a smart contract before and after any operation in a transaction that changes its state. Proof-of-concept implementations show that this solution can provide a detection and prevention mechanism against reentrancy attacks during the execution of any smart contract.

Keywords: blockchain technology, Ethereum, cybersecurity, smart contract, reentrancy vulnerability

OPEN ACCESS

Edited by:
Yannick Chevalier,
Université Toulouse III Paul Sabatier,
France

Reviewed by:
Damián López,
Universitat Politècnica de València,
Spain
Eike Ritter,
University of Birmingham,
United Kingdom

****Correspondence:***
Ayman Alkhalifah
aymankaa@gmail.com

Specialty section:
This article was submitted to
Computer Security,
a section of the journal
Frontiers in Computer Science

Received: *25 August 2020*
Accepted: *04 January 2021*
Published: *17 February 2021*

Citation:
Alkhalifah A, Ng A, Watters PA and Kayes ASM (2021) A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks.
Front. Comput. Sci. 3:598780.
doi: 10.3389/fcomp.2021.598780

INTRODUCTION

Since 2015, when Ethereum smart contracts were introduced, there have been several incidents in which the operation of smart contracts that held an amount of Ether resulted in conflicts or issues (Alkhalifah et al., 2019). Two of these incidents were caused by reentrancy vulnerability. The first incident was in 2016 when an attacker launched a reentrance attack against the distributed autonomous organization (DAO) smart contract, which resulted in a loss of more than 3,600,000 million Ethers worth more than US$ 60 million at that time; the Ether market plunged, and the incident caused a hard fork leading to two versions of the Ethereum blockchain. The second incident was in 2018 when an adversary stole more than 165 Ethers, which is worth almost US$ 40,000, from SpankChain due to reentrancy vulnerability in the smart contract of the network's payment channel (Alkhalifah et al., 2020). Despite these incidents, the popularity of smart contracts is growing; however, they are also becoming more attractive targets for adversaries.

Smart contracts are one of the most used attack vectors to Ethereum because they are like any other executable applications that operate on computers. Nevertheless, smart contracts are more sensitive in terms of cybersecurity due to the following factors:

- Smart contracts in Ethereum blockchain are considered a new platform; therefore, the coding practices of smart contracts are not yet mature (Hung et al., 2019).
- The smart contract can be associated with a digital fortune that might be worth millions of dollars (Schrans et al., 2018).
- The smart contracts operate on top of a blockchain that is immutable. Thus, once the smart contracts are deployed on the blockchain, it can possibly be impractical and intensely complex to modify them even if they contain flaws (the "code is law" concept) (Madnick 2020).

Is it true that the "code is law" in Ethereum smart contracts? Many countermeasures are aimed to detect security flaws during the development stage since the contracts operate on top of immutable technology. However, smart contract developers need to act to address these flaws during the execution stage and not only rely on the development stage. The question is how we can secure a smart contract during the execution stage, even though it is immutable and contains flaws.

This article is organized as follows. In the *Related Work and the Limitations of Current Solutions* section, we analyze the related literature into two categories: single-function and cross-function reentrancy attacks. We summarize the limitations of existing solutions while preventing vulnerability exploitation before the deployment of smart contracts. We propose a solution along with a prevention mechanism to protect smart contracts and a detection technique to identify attackers in *The Proposed Solution* section. The *Implementation of the Solution* and *Proof of Concept* sections describe the three approaches of our implementation architecture and the proof-of-concept result, which shows that the proposed solution can be utilized against reentrancy attacks during the execution of any smart contract. We finally summarize the findings and provide the future research directions in the *Conclusion and Future Research* section.

RELATED WORK AND THE LIMITATIONS OF CURRENT SOLUTIONS

Reentrancy attacks are one of the common threats in Ethereum blockchain, which are associated with the Solidity programming language. The attacks occur when an adversary leverages an external call of a smart contract by forcing the contract to execute additional code by utilizing a fallback function to call back to itself.

There are two types of reentrancy attacks (Samreen and Alalfi 2020): single-function and cross-function attack. A single-function attack occurs when the adversary attempts to recursively call the vulnerable function. A cross-function attack occurs when the target function state is shared with another function that the adversary desires to exploit.

Current Defensive Methods

There are different methods used to protect smart contracts from reentrancy vulnerability. These proactive methods, which are utilized before the deployment of the smart contracts, are vulnerability-detection tools for Ethereum smart contracts, security based on programming languages, and security based on the development of smart contracts.

Smart Contracts Vulnerabilities Detection Tools

There are several detection tools for smart contract vulnerabilities that can detect reentrancy vulnerability. The following sections will briefly introduce the detection tools that can discover the reentrancy vulnerability.

SmartCheck

SmartCheck is a code analysis tool that detects code issues in Solidity. The source code written by Solidity is translated into an XML-based intermediate representation. After that, SmartCheck checks the output against XPath patterns (Tikhomirov et al., 2018). SmartCheck automatically checks for atrocious coding practices and vulnerabilities by highlighting them and providing a vulnerability explanation with a suggested solution to avoid cybersecurity issues (Dika and Nowostawski, 2018).

Remix

Remix is a web-based IDE for writing and debugging smart contracts by utilizing high-level languages such as Solidity and Vyper. Remix identifies the possible vulnerable coding pattern and minimizes coding mistakes. It can identify several vulnerabilities such as reentrancy, timestamp dependence, and gas-costly patterns vulnerabilities (Dika and Nowostawski, 2018).

Oyente

Oyente is a symbolic execution tool that helps smart contract developers to detect possible vulnerabilities as a mitigation technique before the deployment. Oyente pursues the smart contracts execution paradigm in Ethereum blockchain and directly operates on the EVM bytecode without the need to access the high-level source code. The tool is open-source and is available for public use (Luu et al., 2016; Lee, 2018).

Mythril

Mythril is a security analysis tool to analyze smart contracts' security issues in the Ethereum blockchain. It provides a different analysis of vulnerabilities in smart contracts based on symbolic code execution (Prechtel et al., 2019). Mythril works with EVM bytecode to detect cybersecurity vulnerabilities in smart contracts that are developed for EVM-compatible blockchains such as Ethereum and Tron. It uses taint analysis and symbolic execution to detect several vulnerabilities such as reentrancy and unprotected functions vulnerabilities (Zhang et al., 2019).

Securify

Securify is a cybersecurity analyzer for smart contracts in the Ethereum blockchain that is fully automated and scalable and categorizes the contract behaviors into safe or unsafe based on a provided property. There are two steps in Securify analysis: the first is extracting semantic information from the code by symbolically analyzing the dependency graph of the smart contract; the second step is checking violation and compliance patterns that set the conditions to identify whether a property holds or not (Tsankov et al., 2018).

F Framework*

F* framework is a verification method based on F* language offered by Microsoft Research. The smart contract is checked to see if it is correct by translating the code written in Solidity to F* language. Since the smart contracts' binary codes are available on the Ethereum network, whereas the source code is hard to obtain, the binary files on Ethereum blockchain are decompiled to F* language to identify possible vulnerabilities such as exception disorders and reentrancy vulnerabilities (Liu and Liu, 2019).

Security Based on Programming Languages

Several high-level programming languages are introduced to develop smart contracts securely. For instance, Obsidian is a state-oriented language that follows the states of the smart contracts to avoid reentrancy vulnerability and treats Ether as a linear resource to allow the compiler to track financial information. Currently, Obsidian is not ready for general use and still in the development stage (Coblenz, 2017).

Another example is Vyper that is also a high-level programming language that includes other new functionalities not supported by Solidity and eliminates some of the Solidity features. It includes new functionalities such as overflow checking and bounds, and it eliminates features such as modifiers and recursive calling. Vyper helps developers to avoid vulnerabilities such as integer underflow and overflow vulnerability and denial-of-service (DOS) with unbounded operations vulnerability (Adrian, 2018).

Security Based on the Development of Smart Contracts

There are a few approaches to enhancing the programming model of smart contracts to aid developers in mitigating or avoiding reentrancy vulnerability. One of them, introduced by ConsenSys Diligence, is Ethereum smart contract best practices. This provides fundamental information about security considerations for Solidity programmers. Furthermore, various recommendations are provided to guide smart contract developers on Ethereum to avoid coding issues. Their recommendations are divided into two categories, namely, protocol-specific recommendations and Solidity-specific recommendations. Protocol-specific recommendations apply to any smart contract development on Ethereum to prevent reentrancy vulnerability, such as avoiding state changes after external calls. The other recommendations are the Solidity-specific recommendations, which might be informative for smart contract developers in other languages to prevent reentrancy attacks, such as using modifiers only for assertions (Chen et al., 2020).

Limitations of the Current Solutions

These solutions are generally based on a predefined specific pattern; when this pattern is detected, the vulnerability in the smart contract code is then detected.

The following limitations were found:

- Detection tools that detect reentrancy vulnerability analyze the smart contract code based on predefined attack patterns, and if the patterns match any part in the code, then the tools discover the vulnerability. Thus, these approaches mainly rely on complete patterns and the specific quality of these patterns.
- The patterns these solutions rely on are based on the observation of the previous attacks and known vulnerabilities, which makes them limited and difficult to generalize.
- All the solutions are only applicable before the deployment of smart contracts. This means once the smart contract is deployed on the Ethereum network, these solutions cannot prevent reentrancy attacks and cannot detect the attacker.
- If a new reentrancy pattern is introduced after the deployment of the smart contracts, these solutions need to be updated; otherwise, they will not be able to detect the new attack patterns.

THE PROPOSED SOLUTION

We have analyzed the root cause of reentrancy attack on the lack of integrity checking on smart contract balance and proposed a solution to overcome these limitations by providing a prevention technique to protect the smart contract and a detection technique to detect the attacker that is not based on any pattern and can be utilized after the deployment of the smart contract. This solution can differentiate between honest and malicious transactions, can be implemented within several approaches, and can be utilized on the current Ethereum platform.

In any smart contract that manages a fund for various participants, two values maintain the funds in the smart contract. The first value is maintained by the protocol layer, which is the contract *balance* represented in Solidity as *address(this).balance*, whereas the second value is maintained by the application layer, which is usually represented in Solidity as *balances[ParticipantAddress]*, which maintains the balance of each participant in the smart contract. The contract balance and the total balance of all participants are not always the same; however, when any smart contract is initiated, the difference between them must always be the same after and before any operation in the smart contract that changes the state of the smart contract in order to protect the funds in the smart contract. This is because adversaries who want to launch a reentrancy attack are aiming to trick the smart contract in the application layer by decreasing the value that is maintained by the protocol layer and at the same time keeping the value that is maintained by the application layer as it is. The attackers do this because they can manipulate the smart contract in the application layer, but they cannot manipulate the value [*address(this).balance*] maintained by the protocol layer because it is secured by the miners who maintain it. The only way the attacker can carry out such an attack is to trick the smart contract in the application layer, which will lead to change the difference between these two values, and if the attacker succeeds in this, then the attacker will be able to

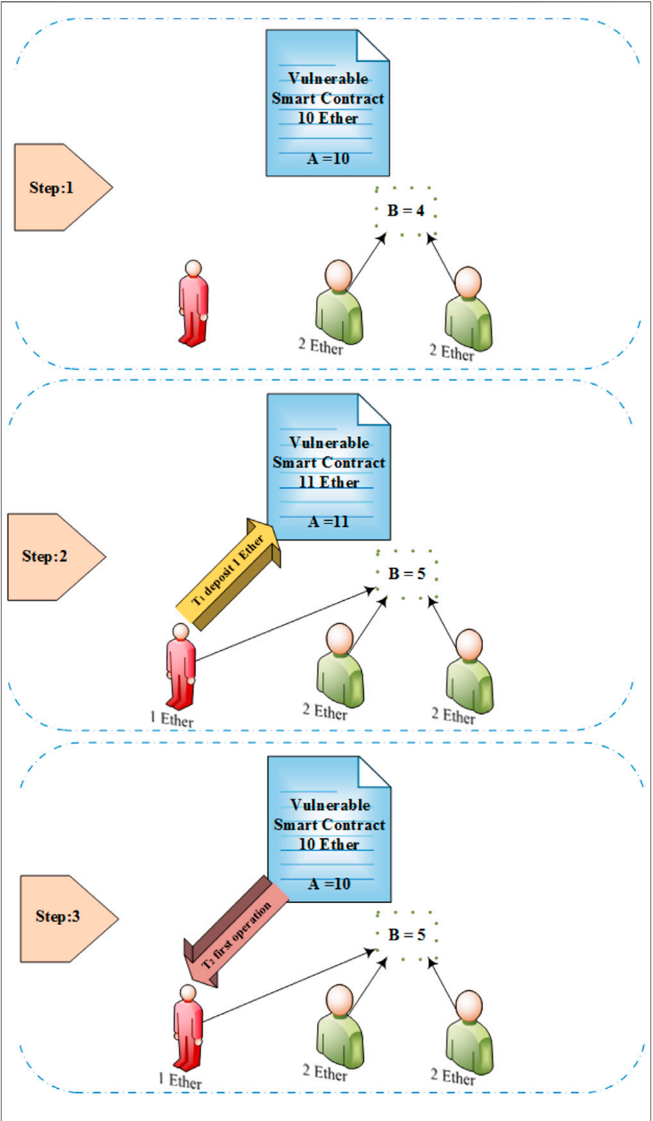


FIGURE 1 | Reentrancy attack scenario.

steal the funds and the smart contract will not be aware of the attack and will not be able to stop the attacker.

This solution is based on these two values, namely, the contract balance (noted as a before the operation and a' after the operation) and the total balance of all participants in the smart contract (noted as b before operation and b' after the operation). The solution simply calculates the difference between a and b before and a' and b' after each operation within a transaction in any smart contract. The transaction is noted as T and the difference before the operation is noted as x, whereas the difference after the operation is noted as x'. If the difference is the same ($x = x'$), then the operation is legitimate; otherwise, the operation is malicious. For instance, suppose that a smart contract has a reentrancy vulnerability and has 10 Ethers in its balance ($a = 10$) and three participants, where two are honest and one is a malicious user who tries to exploit the vulnerability. Each of the honest participants has 2 Ethers in their balance; therefore, the total balance for all participants is 4 ($b = 4$). The difference, in this case, is 6 ($x_1 = 6$). The malicious participant conducts a reentrancy attack. The adversary sends T_1 to deposit 1 Ether to the contract; therefore, $a' = 11$, $b' = 5$, $x_1 = 6$, and $x_1' = 6$. Since $x_1 = x_1'$, this transaction is not malicious. After that, the adversary sends T_2 to withdraw 1 Ether, utilizing the recursive function to exploit the reentrancy vulnerability. After the first operation in T_2 and before the second operation starts, the values will be $a = 11$, $b = 5$, $a' = 10$, $b' = 5$, $x_1 = 6$, and $x_1' = 5$. $x_1 \neq x_1'$ because the execution did not reach the statement that decrements the balance of malicious user; therefore, a' will be decreased by 1; however, the b' value will still be the same, as shown in **Figure 1**. This operation will be blocked, and the transaction will be considered a malicious transaction; therefore, the attacker's address will be stored by the solution.

Our solution formula is as follows.

$\forall\ O \in T$: (O is valid) $\Leftrightarrow$ ($x = x'$),
where
T = a transaction,
O = an operation in the transaction that changes the smart contract state,
x = a − b,
x′ = a′ − b′,
a = contract balance before O,
$b = \sum_{i=1}^{n} p_i$ before O,
a' = contract balance after O,
$b' = \sum_{i=1}^{n} p_i$ after O,
p_i = the balance of participant i in the contract, and
n = the number of participants in the contract.

IMPLEMENTATION OF THE SOLUTION

There are three approaches to implementing the architecture of the solution, as depicted in **Figure 2**. This solution can be implemented on the dApp layer (Approach 1), or on the blockchain layer as an independent smart contract (Approach 2) to control one or more smart contracts' behavior, or within a smart contract (Approach 3).

In Approach 1, the dApp layer is considered as a checkpoint that receives data from the smart contract. Based on these data, the dApp compares the difference between the stored value of the smart contract balance and the total balance of all participants and the difference between the current contract balance and the current total of all participants. If they match, the dApp sends the decision "true" to allow the operation in the transaction to be completed; otherwise, it sends "false" to block the operation of the transaction and notifies the smart contract owner by storing the attacker's address.

Approach 2 is similar to Approach 1; however, another smart contract will play the dApp checkpoint role. All malicious operations in the transactions will be blocked and the smart contract owner will also be notified by storing the attacker's address in the contract.

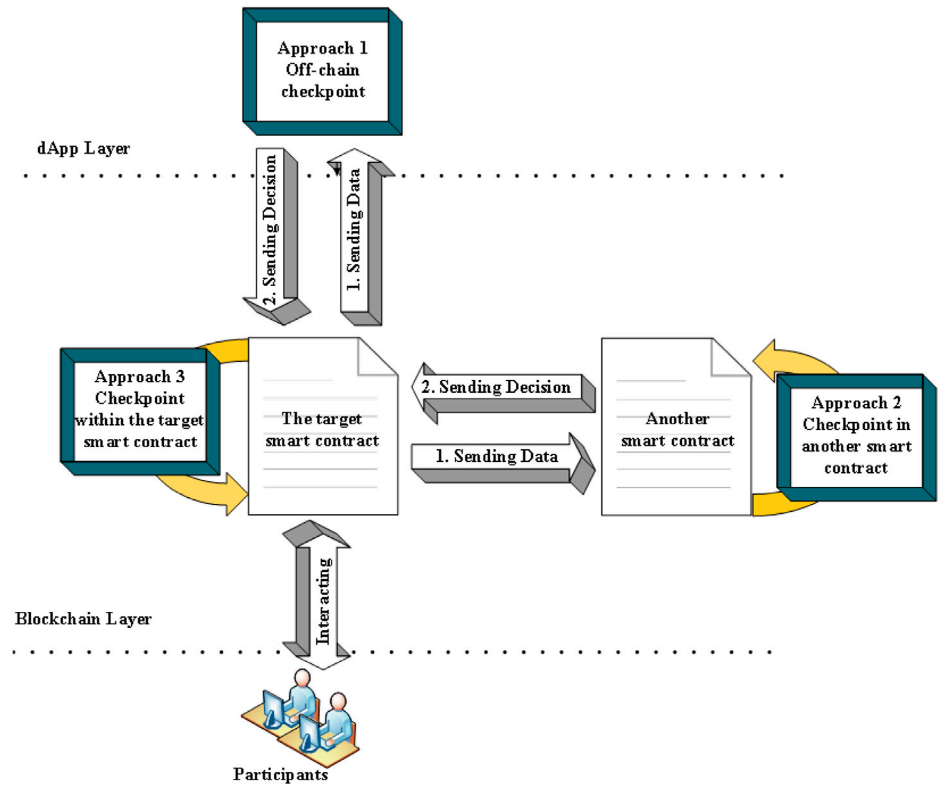


FIGURE 2 | Three approaches to implementing the architecture of the solution.

Approach 3 works within the smart contract itself, monitoring the operations that change the state of the smart contract; and if any malicious operation occurs, the smart contract will block the operation and the attacker address will be stored to notify the owner.

Figure 3 illustrates the solution data flow which is applicable in all three approaches.

The first three steps in the data flow will happen during the initialization of the smart contract that needs the protection. The solution should obtain the smart contract balance and the total balance of all smart contracts' participants. The difference between these values should be calculated in the third step. All participants can interact with the smart contract in the fourth step and during the execution, the solution continuously monitors the smart contract balance and all participants balance before and after each operation that changes the smart contract state. In the seventh step, the solution calculates the difference again and compares the result with the stored result. If the results are equal, the value of x is changed to the value of x' and the execution completes; otherwise, the execution is blocked and the smart contract owner will be notified by storing the address of the attacker in the contract. **Figure 4** shows an example of a UML class diagram of Approach 3, which can be applied for a bank smart contract.

Monitoring the contract balance and all the participants' balances to prevent reentrancy vulnerability and detect the attacker can be implemented in various ways. This section will provide an example of the implementation of Approach 3 based on the previous UML class diagram, which is written by using the Solidity programming language as shown in **Figure 5**. There are six variables and eight functions with the constructor. Four variables are utilized by the solution: *participantsLiquidity*, *beforeOperation*, *afterOperation*, and *attacker*. The reentrancy vulnerability is stated in the code in line (46) that may cause different invocations for different functions, which will be illustrated in the test scenarios in the next chapter. The contract owner is the only one who can retrieve the address of the attacker because of the modifier in line (20).

PROOF OF CONCEPT

The Testing Environment

Remix IDE was used to host the test environment and to compile, deploy, debug, and test the solution. This test utilized the JavaScript VM environment, which emulates a real blockchain, to execute all the test transactions. All the smart contracts that were used in this test were written in the Solidity programming language. Two attack case studies were conducted: a single-function and a cross-function reentrancy attack. The single-function reentrancy attack case study contained three smart contracts, the *Bank{}*, *Attacker{}*, and *BankWithoutSolution{}* contracts. The *Bank{}* and *BankWithoutSolution{}* contracts included a reentrancy vulnerability; however, the *Bank{}*

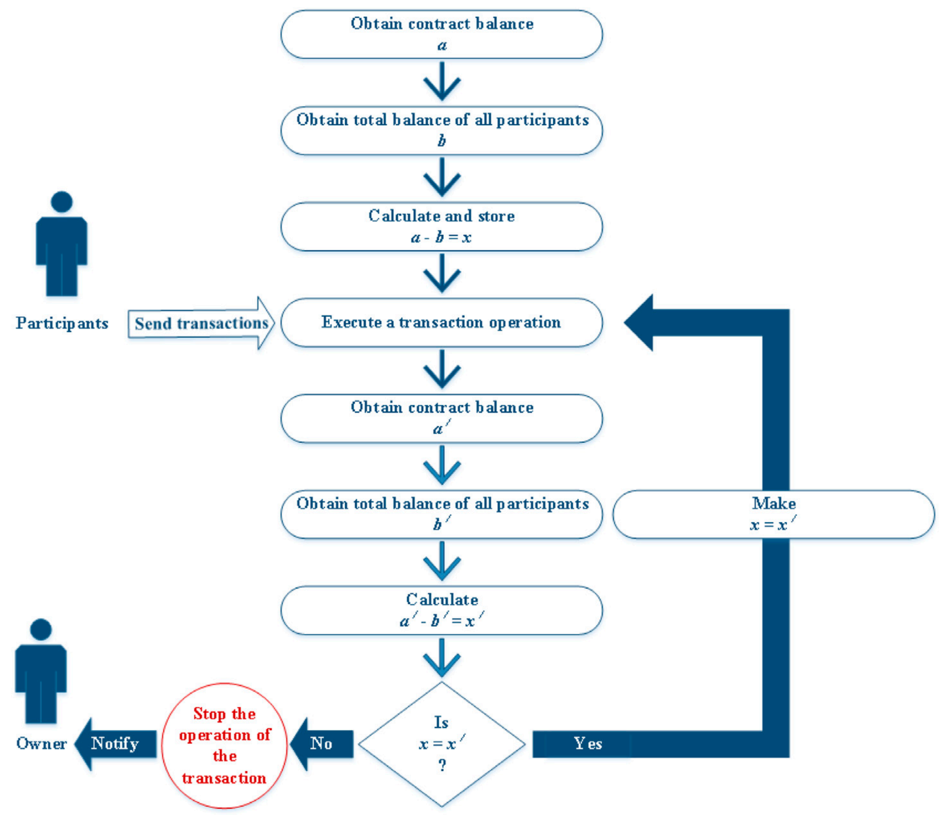

Obtain contract balance a
Obtain total balance of all participants b
Calculate and store $a - b = x$
Participants
Send transactions
Execute a transaction operation
Obtain contract balance a'
Obtain total balance of all participants b'
Make $x = x'$
Calculate $a' - b' = x'$
Owner
Notify
Stop the operation of the transaction
No
Is $x = x'$?
Yes

FIGURE 3 | The solution data flow.

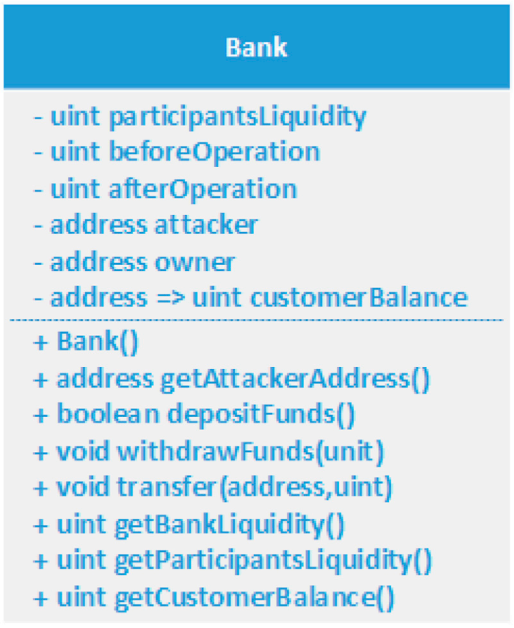
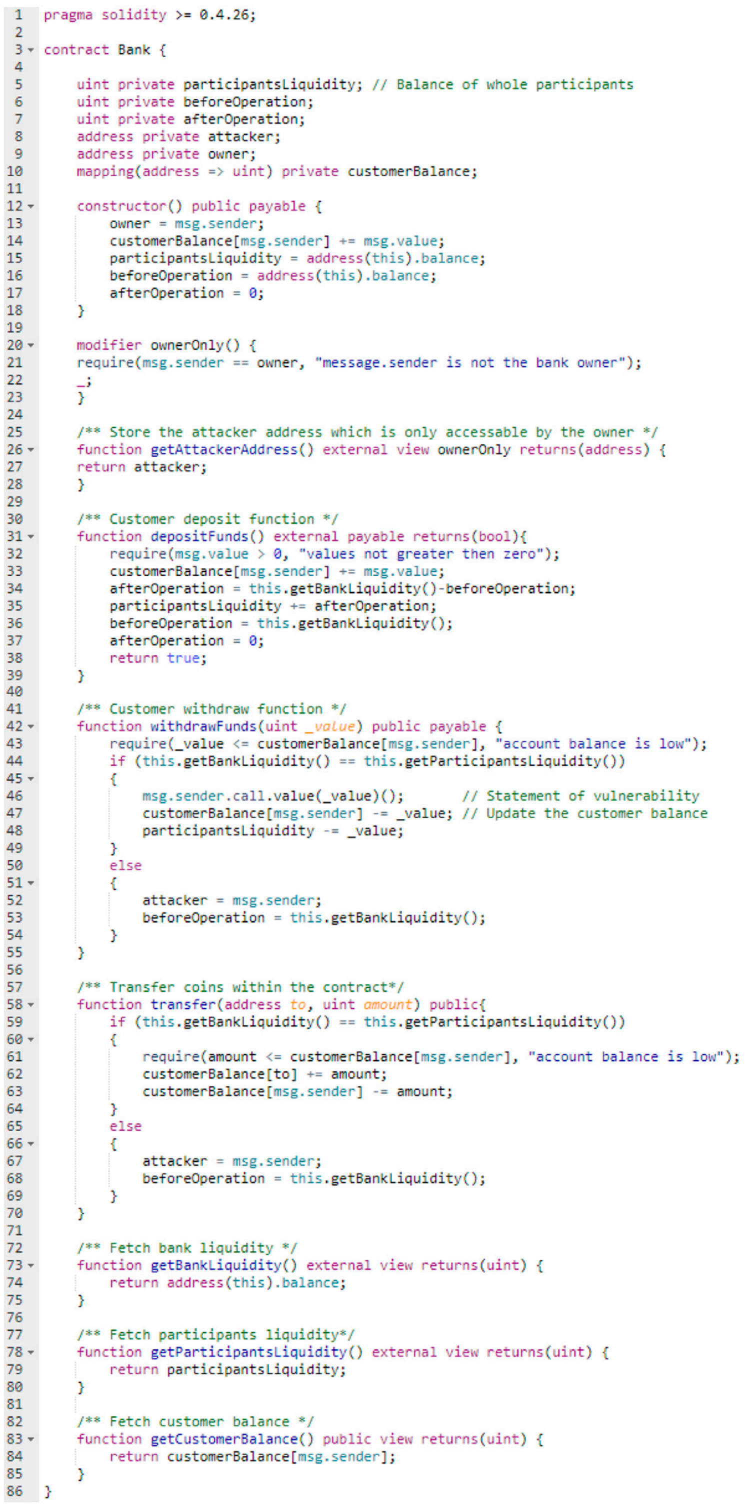

Bank
- uint participantsLiquidity
- uint beforeOperation
- uint afterOperation
- address attacker
- address owner
- address => uint customerBalance
+ Bank()
+ address getAttackerAddress()
+ boolean depositFunds()
+ void withdrawFunds(unit)
+ void transfer(address,uint)
+ uint getBankLiquidity()
+ uint getParticipantsLiquidity()
+ uint getCustomerBalance()

FIGURE 4 | UML class diagram example for the solution applied to a bank contract.

contract included the solution as shown in **Figure 5** in Section *Implementation of the Solution,* whereas the *BankWithoutSolution{}* contract did not use the solution. The *Attacker{}* contract included the malicious code that conducted the attacks. The second case study is the cross-function reentrancy attack case study, which included four smart contracts *Bank{}*, *Attacker1{}*, *BankWithoutSolution{}*, and *Attacker2{}* contracts. The *Bank{}* and *BankWithoutSolution{}* contracts were the same as those in the first case study and the *Attacker1{}* contract was utilized to launch the attack, while the *Attacker2{}* contract was utilized to receive the stolen coin. Both case studies will be illustrated in detail in the following sections.

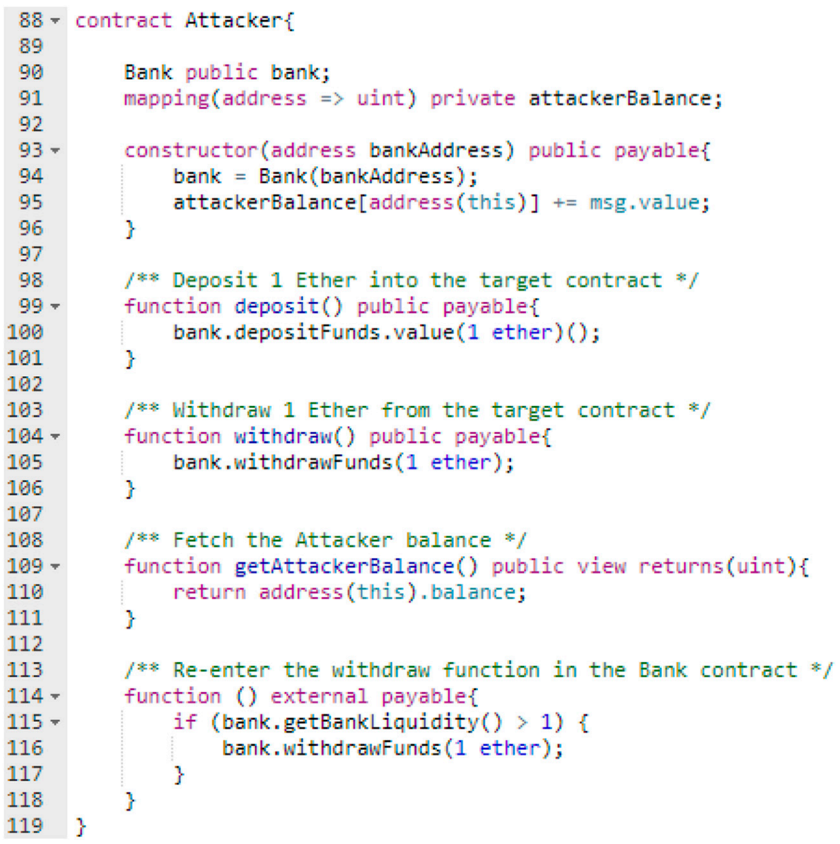
The Testing Scenario

Each of the two case studies consists of two test scenarios, which are firstly conducting the attack without the solution and secondly conducting the attack with the solution. The *Bank{}* and *BankWithoutSolution{}* contracts were funded by an individual account by 10 Ethers during the deployment. The first test scenario in each attack case study was conducted to see if the attacker was able to steal coins from the *BankWithoutSolution{}* contract and the second test scenario was conducted to see if the solution was able to detect and prevent the reentrancy attack from the *Bank{}* contract.



```
pragma solidity >= 0.4.26;

contract Bank {

    uint private participantsLiquidity; // Balance of whole participants
    uint private beforeOperation;
    uint private afterOperation;
    address private attacker;
    address private owner;
    mapping(address => uint) private customerBalance;

    constructor() public payable {
        owner = msg.sender;
        customerBalance[msg.sender] += msg.value;
        participantsLiquidity = address(this).balance;
        beforeOperation = address(this).balance;
        afterOperation = 0;
    }

    modifier ownerOnly() {
    require(msg.sender == owner, "message.sender is not the bank owner");
    _;
    }

    /** Store the attacker address which is only accessable by the owner */
    function getAttackerAddress() external view ownerOnly returns(address) {
    return attacker;
    }

    /** Customer deposit function */
    function depositFunds() external payable returns(bool){
        require(msg.value > 0, "values not greater then zero");
        customerBalance[msg.sender] += msg.value;
        afterOperation = this.getBankLiquidity()-beforeOperation;
        participantsLiquidity += afterOperation;
        beforeOperation = this.getBankLiquidity();
        afterOperation = 0;
        return true;
    }

    /** Customer withdraw function */
    function withdrawFunds(uint _value) public payable {
        require(_value <= customerBalance[msg.sender], "account balance is low");
        if (this.getBankLiquidity() == this.getParticipantsLiquidity())
        {
            msg.sender.call.value(_value)();       // Statement of vulnerability
            customerBalance[msg.sender] -= _value; // Update the customer balance
            participantsLiquidity -= _value;
        }
        else
        {
            attacker = msg.sender;
            beforeOperation = this.getBankLiquidity();
        }
    }

    /** Transfer coins within the contract*/
    function transfer(address to, uint amount) public{
        if (this.getBankLiquidity() == this.getParticipantsLiquidity())
        {
            require(amount <= customerBalance[msg.sender], "account balance is low");
            customerBalance[to] += amount;
            customerBalance[msg.sender] -= amount;
        }
        else
        {
            attacker = msg.sender;
            beforeOperation = this.getBankLiquidity();
        }
    }

    /** Fetch bank liquidity */
    function getBankLiquidity() external view returns(uint) {
        return address(this).balance;
    }

    /** Fetch participants liquidity*/
    function getParticipantsLiquidity() external view returns(uint) {
        return participantsLiquidity;
    }

    /** Fetch customer balance */
    function getCustomerBalance() public view returns(uint) {
        return customerBalance[msg.sender];
    }
}
```

FIGURE 5 | The solution is implemented in the code of the bank smart contract.



```
88  contract Attacker{
89
90      Bank public bank;
91      mapping(address => uint) private attackerBalance;
92
93      constructor(address bankAddress) public payable{
94          bank = Bank(bankAddress);
95          attackerBalance[address(this)] += msg.value;
96      }
97
98      /** Deposit 1 Ether into the target contract */
99      function deposit() public payable{
100         bank.depositFunds.value(1 ether)();
101     }
102
103     /** Withdraw 1 Ether from the target contract */
104     function withdraw() public payable{
105         bank.withdrawFunds(1 ether);
106     }
107
108     /** Fetch the Attacker balance */
109     function getAttackerBalance() public view returns(uint){
110         return address(this).balance;
111     }
112
113     /** Re-enter the withdraw function in the Bank contract */
114     function () external payable{
115         if (bank.getBankLiquidity() > 1) {
116             bank.withdrawFunds(1 ether);
117         }
118     }
119 }
```

FIGURE 6 | The *Attacker{}* contract.

Single-Function Reentrancy Attack Case Study

The *Attacker{}* contract as shown in **Figure 6** will be used in the following scenarios. The *Attacker{}* contract code in the first scenario will be changed in lines (90) and (94) from *Bank* to *BankWithoutSolution*.

First Test Scenario: Single-Function Attack (Without the Solution)

The first test scenario conducted a single-function reentrancy attack against *BankWithoutSolution{}* contract. The *BankWithoutSolution{}* contract code is shown in **Figure 7**, which contained the vulnerability in line (23).

The attack sequence diagram for the first test scenario in the single-function reentrancy attack case study is shown in **Figure 8**. All the transactions and calls involved in the first test scenario are shown in **Supplementary Appendix 1**.

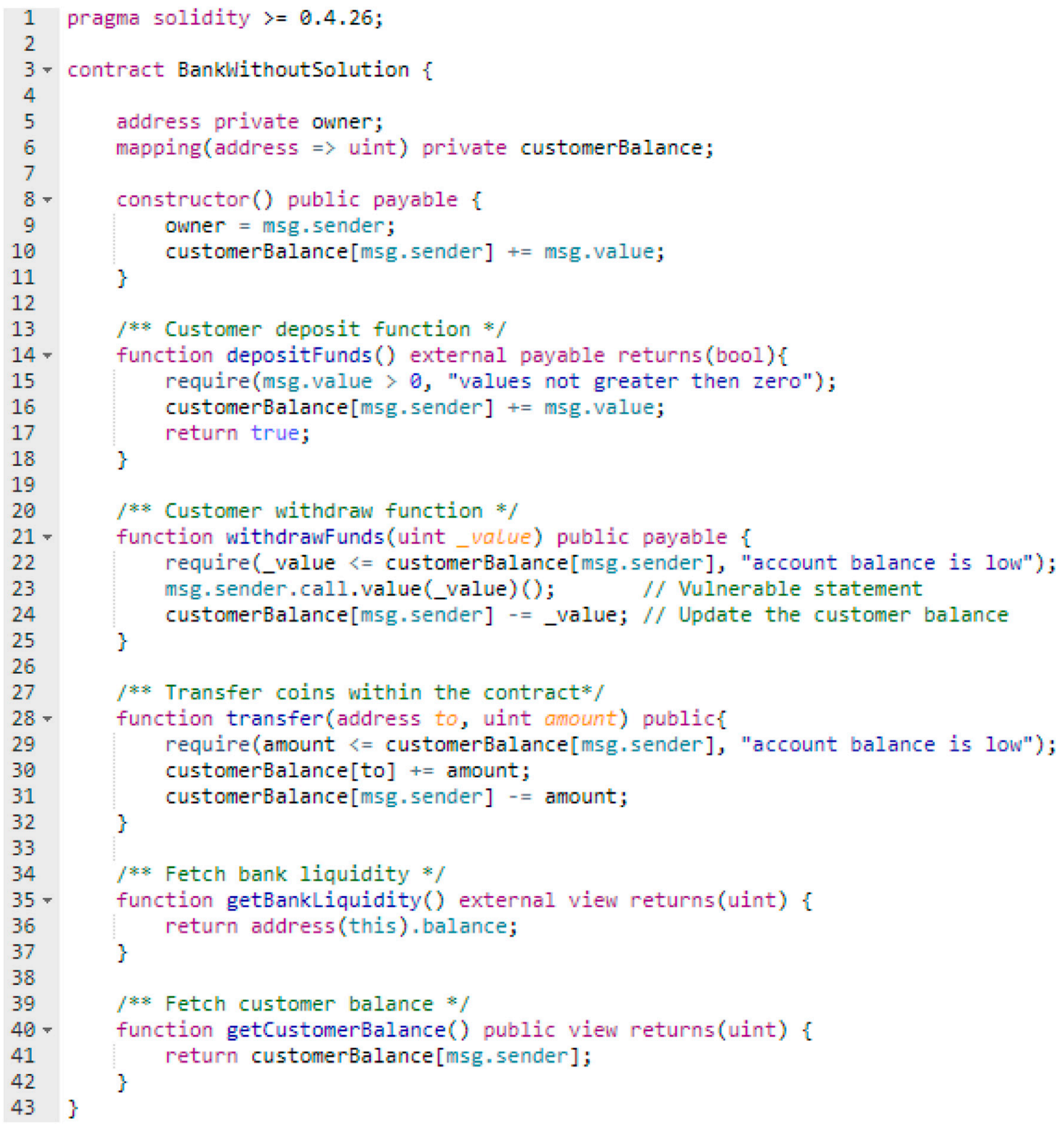
The scenario steps were as follows:

1. Deploying the *BankWithoutSolution{}* contract with funds equal to 10 Ethers.
2. Deploying the *Attacker{}* contract by passing the *BankWithoutSolution{}* contract address as a parameter in the *Attacker{}* constructor and with funds equal to 1 Ether.
3. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check the balance, which is equal to1 Ether.
4. Depositing 1 Ether from the *Attacker{}* contract to the *BankWithoutSolution{}* contract.
5. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check that the balance is equal to 0 Ethers.
6. Calling the *getBankLiquidity()* function in the *BankWithoutSolution{}* contract to check that the balance is equal to 11 Ethers.
7. Conducting the single-function reentrancy attack by calling the withdraw function in the *Attacker{}* contract.
8. Calling the *getBankLiquidity()* function in the *BankWithoutSolution{}* contract to check the balance, which is equal to 0 Ethers.
9. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check the balance, which is equal to 11 Ethers.

In the first scenario, the single-function reentrancy attack succeeded in leveraging the reentrancy vulnerability and the attacker stole all the *BankWithoutSolution{}* contract funds.

Second Test Scenario: Single-Function Attack (With the Solution)

The second test scenario conducts a single-function reentrancy attack on the *Bank{}* contract. The *Bank{}* code is shown in



```
pragma solidity >= 0.4.26;

contract BankWithoutSolution {

    address private owner;
    mapping(address => uint) private customerBalance;

    constructor() public payable {
        owner = msg.sender;
        customerBalance[msg.sender] += msg.value;
    }

    /** Customer deposit function */
    function depositFunds() external payable returns(bool){
        require(msg.value > 0, "values not greater then zero");
        customerBalance[msg.sender] += msg.value;
        return true;
    }

    /** Customer withdraw function */
    function withdrawFunds(uint _value) public payable {
        require(_value <= customerBalance[msg.sender], "account balance is low");
        msg.sender.call.value(_value)();       // Vulnerable statement
        customerBalance[msg.sender] -= _value; // Update the customer balance
    }

    /** Transfer coins within the contract*/
    function transfer(address to, uint amount) public{
        require(amount <= customerBalance[msg.sender], "account balance is low");
        customerBalance[to] += amount;
        customerBalance[msg.sender] -= amount;
    }

    /** Fetch bank liquidity */
    function getBankLiquidity() external view returns(uint) {
        return address(this).balance;
    }

    /** Fetch customer balance */
    function getCustomerBalance() public view returns(uint) {
        return customerBalance[msg.sender];
    }
}
```

FIGURE 7 | The *BankWithoutSolution{}* contract.

Figure 5 in Section *Implementation of the Solution*, which contains the vulnerability in line (46). The attack sequence diagram for the second test scenario in the single-function reentrancy attack case study is shown in **Figure 9**. All of the transactions and calls involved in the second test scenario are shown in **Supplementary Appendix 2**.

The scenario steps were as follows:

1. Deploying the *Bank{}* contract with funds equal to 10 Ethers.
2. Deploying the *Attacker{}* contract by passing the *Bank{}* contract address as a parameter in the *Attacker{}* constructor and with funds equal to 1 Ether.
3. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check the balance, which is equal to 1 Ether.
4. Depositing 1 Ether from the *Attacker{}* contract to the *Bank{}* contract.
5. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check that the balance is equal to 0 Ethers.
6. Calling the *getBankLiquidity()* function in the *Bank{}* contract to check that the balance is equal to 11 Ethers.
7. Conducting the single-function reentrancy attack by calling the withdraw function in the *Attacker{}* contract.
8. Calling the *getBankLiquidity()* function in the *Bank{}* contract to check the balance, which is equal to 10 Ethers.
9. Calling the *getAttackerBalance()* function in the *Attacker{}* contract to check the balance, which is equal to 1 Ether.

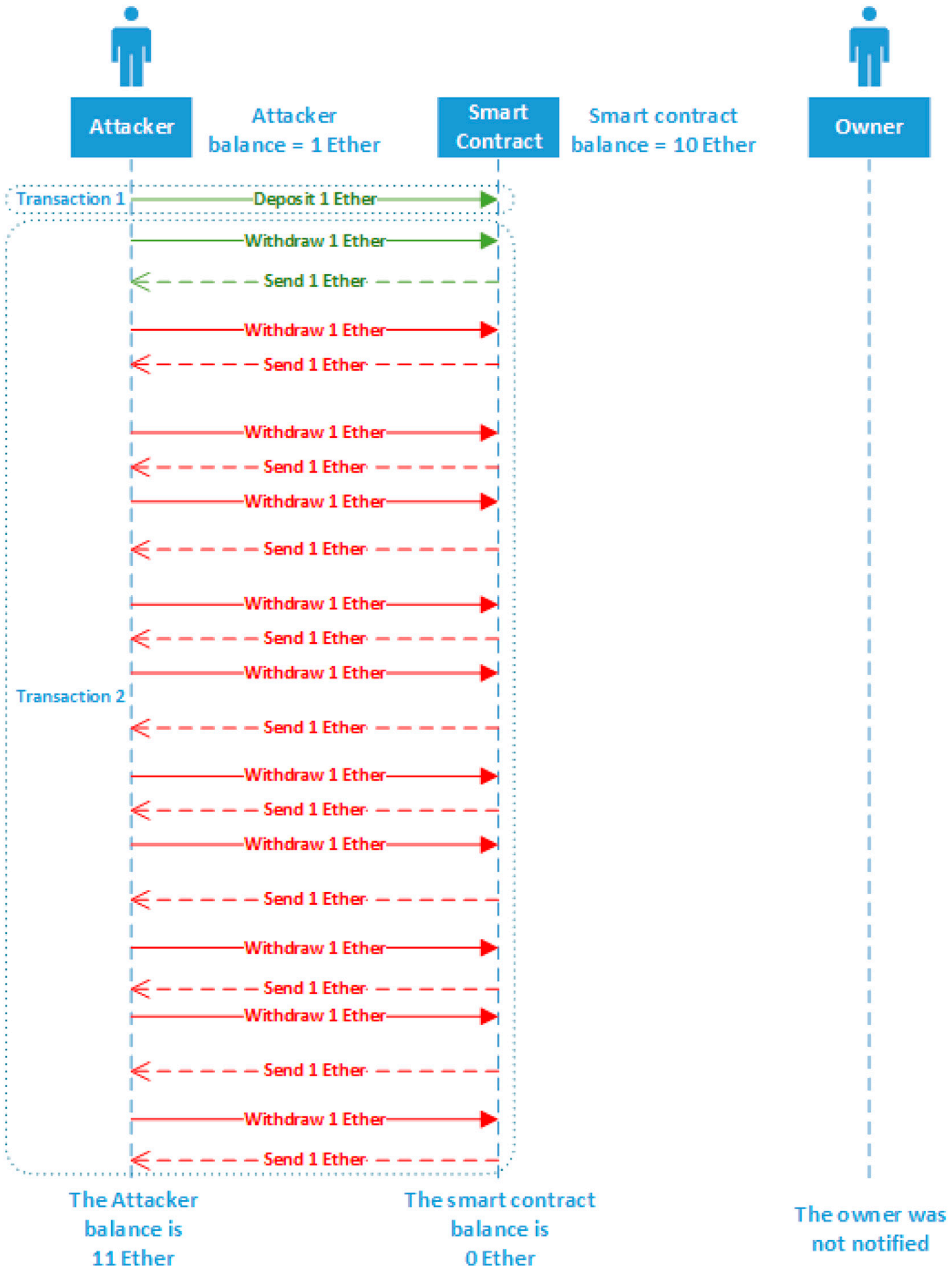

Attacker
Attacker balance = 1 Ether
Smart Contract
Smart contract balance = 10 Ether
Owner
Transaction 1
Deposit 1 Ether
Withdraw 1 Ether
Send 1 Ether
Transaction 2
The Attacker balance is 11 Ether
The smart contract balance is 0 Ether
The owner was not notified

FIGURE 8 | UML sequence diagram for the first scenario in the first case study.

10. Calling the *getAttackerAddress()* in the *Bank{}* contract from the owner account to retrieve the attacker address.

In the second scenario, the single-function reentrancy attack failed to leverage the reentrancy vulnerability and the *Attacker{}* address was stored in the *Bank{}* contract.

Cross-Function Reentrancy Attack Case Study

The *Attacker1{}* and *Attacker2{}* contracts as shown in **Figure 10** were utilized in the following scenarios. The *Attacker1{}* and *Attacker2{}* contracts code in the first scenario were changed in lines (86) and (91) in *Attacker1{}* and line (123) in *Attacker2{}* from *Bank* to *BankWithoutSolution*.

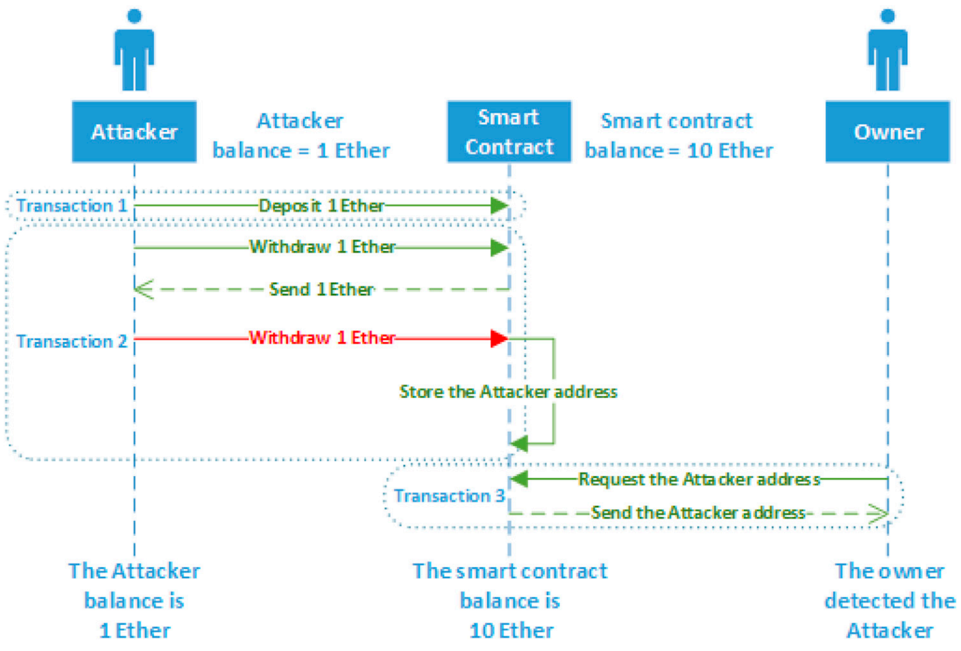
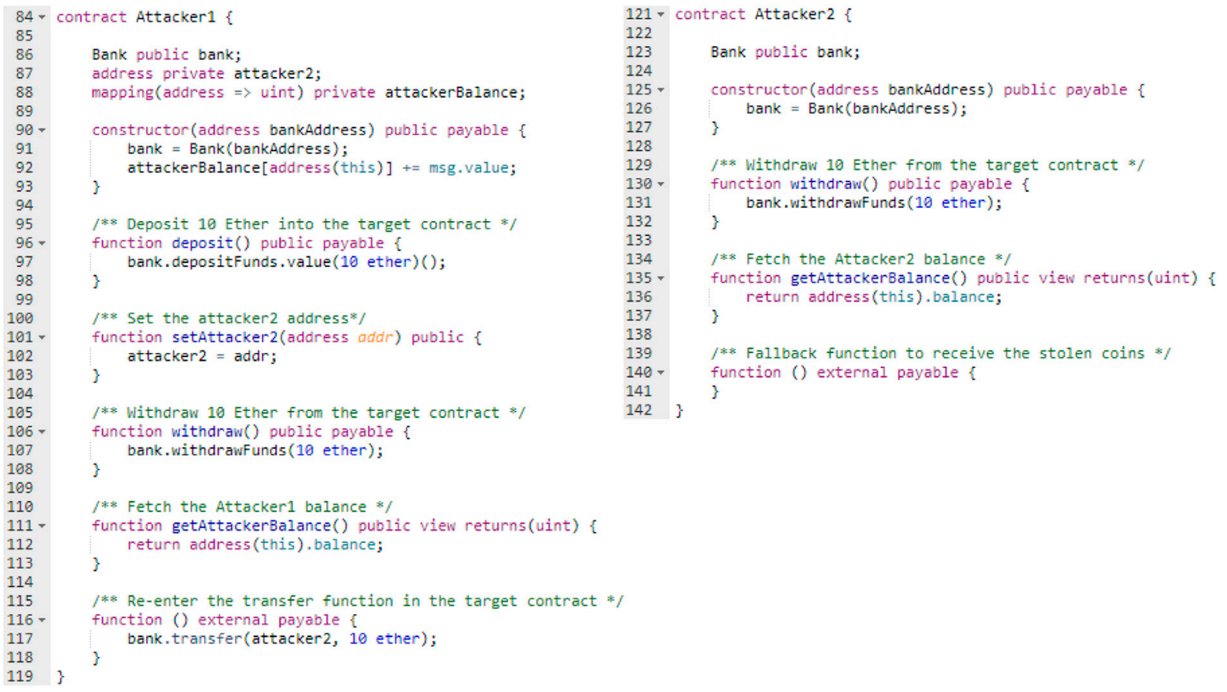

Attacker
Attacker balance = 1 Ether
Smart Contract
Smart contract balance = 10 Ether
Owner
Transaction 1
Deposit 1 Ether
Withdraw 1 Ether
Send 1 Ether
Transaction 2
Withdraw 1 Ether
Store the Attacker address
Transaction 3
Request the Attacker address
Send the Attacker address
The Attacker balance is 1 Ether
The smart contract balance is 10 Ether
The owner detected the Attacker

FIGURE 9 | UML sequence diagram for the second scenario in the first case study.



```
contract Attacker1 {

    Bank public bank;
    address private attacker2;
    mapping(address => uint) private attackerBalance;

    constructor(address bankAddress) public payable {
        bank = Bank(bankAddress);
        attackerBalance[address(this)] += msg.value;
    }

    /** Deposit 10 Ether into the target contract */
    function deposit() public payable {
        bank.depositFunds.value(10 ether)();
    }

    /** Set the attacker2 address*/
    function setAttacker2(address addr) public {
        attacker2 = addr;
    }

    /** Withdraw 10 Ether from the target contract */
    function withdraw() public payable {
        bank.withdrawFunds(10 ether);
    }

    /** Fetch the Attacker1 balance */
    function getAttackerBalance() public view returns(uint) {
        return address(this).balance;
    }

    /** Re-enter the transfer function in the target contract */
    function () external payable {
        bank.transfer(attacker2, 10 ether);
    }
}
```

```
contract Attacker2 {

    Bank public bank;

    constructor(address bankAddress) public payable {
        bank = Bank(bankAddress);
    }

    /** Withdraw 10 Ether from the target contract */
    function withdraw() public payable {
        bank.withdrawFunds(10 ether);
    }

    /** Fetch the Attacker2 balance */
    function getAttackerBalance() public view returns(uint) {
        return address(this).balance;
    }

    /** Fallback function to receive the stolen coins */
    function () external payable {
    }
}
```

FIGURE 10 | The *Attacker1{}* and *Attacker2{}* contracts code.

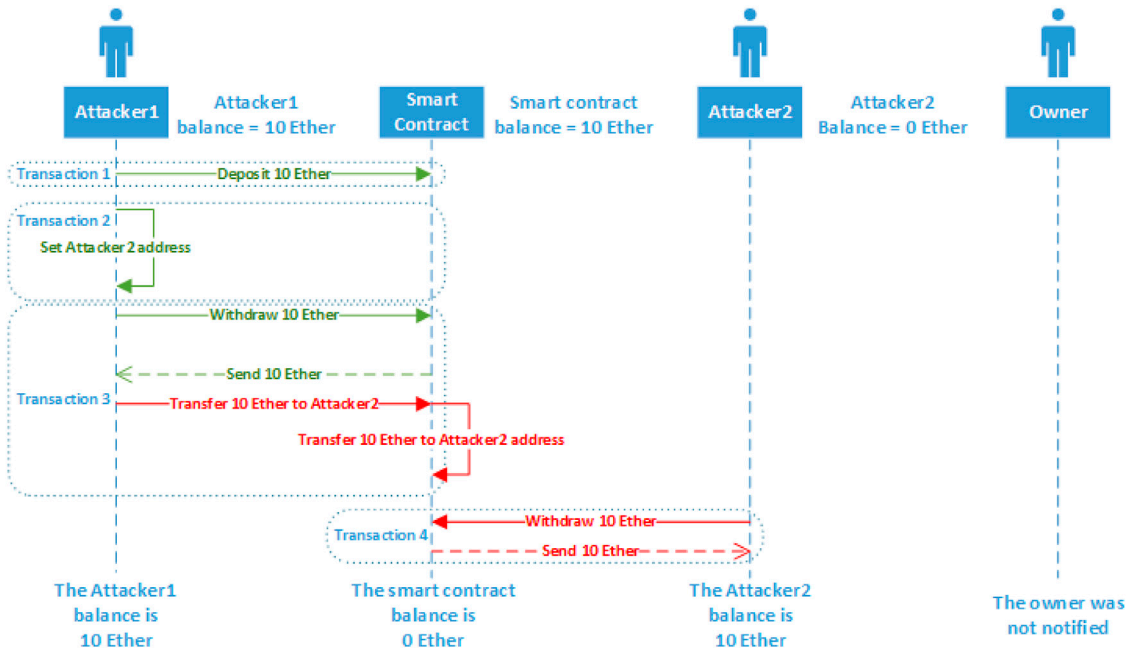

Attacker1
Attacker1 balance = 10 Ether
Smart Contract
Smart contract balance = 10 Ether
Attacker2
Attacker2 Balance = 0 Ether
Owner
Transaction 1
Deposit 10 Ether
Transaction 2
Set Attacker2 address
Withdraw 10 Ether
Send 10 Ether
Transaction 3
Transfer 10 Ether to Attacker2
Transfer 10 Ether to Attacker2 address
Transaction 4
Withdraw 10 Ether
Send 10 Ether
The Attacker1 balance is 10 Ether
The smart contract balance is 0 Ether
The Attacker2 balance is 10 Ether
The owner was not notified

FIGURE 11 | UML sequence diagram for the first scenario in the second case study.

First Test Scenario: Cross-Function Attack (Without the Solution)

The first test scenario launched a cross-function reentrancy attack on the *BankWithoutSolution{}* contract. The *BankWithoutSolution{}* code is shown in **Figure 7**, which contained the vulnerability in line (23). The attack sequence diagram for the first test scenario in the cross-function reentrancy attack case study is shown in **Figure 11**. All the transactions and calls involved in the first test scenario are shown in **Supplementary Appendix 3**.

The scenario steps were as follows:

1. Deploying the *BankWithoutSolution{}* contract with funds equal to 10 Ethers.
2. Deploying the *Attacker1{}* contract by passing the *BankWithoutSolution{}* contract address as a parameter in the *Attacker1{}* constructor and with funds equal to 10 Ethers.
3. Deploying the *Attacker2{}* contract by passing the *BankWithoutSolution{}* contract address as a parameter in the *Attacker2{}* constructor.
4. Calling the *getAttackerBalance()* function in the *Attacker2{}* contract to check the balance, which is equal to 0 Ethers.
5. Depositing 10 Ethers from the *Attacker1{}* contract to the *BankWithoutSolution{}* contract.
6. Calling the *getAttackerBalance()* function in the *Attacker1{}* contract to check that the balance is equal to 0 Ethers.
7. Calling the *getBankLiquidity()* function in the *BankWithoutSolution{}* contract to check that the balance is equal to 20 Ethers.
8. Calling the *setAttacker2()* function in the *Attacker1{}* contract and passing the *Attacker2{}* contract address as a parameter.
9. Conducting the cross-function reentrancy attack by calling the *withdraw()* function in the *Attacker1{}* contract.
10. Calling *withdraw()* function in the *Attacker2{}* contract to steal 10 Ethers.
11. Calling the *getBankLiquidity()* function in the *BankWithoutSolution{}* contract to check the balance, which is equal to 0 Ethers.
12. Calling the *getAttackerBalance()* function in the *Attacker1{}* contract to check the balance, which is equal to 10 Ethers.
13. Calling the *getAttackerBalance()* function in the *Attacker2{}* contract to check the balance, which is equal to 10 Ethers.

In the first scenario, the cross-function reentrancy attack succeeded in leveraging the reentrancy vulnerability, and the *Attacker2{}* contract withdrew 10 Ethers even though the contract did not have any funds in the *BankWithoutSolution{}* contract.

Second Test Scenario: Cross-Function Attack (With the Solution)

The second test scenario conducted a cross-function reentrancy attack against the *Bank{}* contract. The *Bank{}* contract code is shown in **Figure 5** in the *Implementation of the Solution* section, which contained the vulnerability in line (46). The attack sequence diagram for the second test scenario in the cross-function reentrancy attack case study is shown in **Figure 12**. All the transactions and calls involved in the second test scenario are shown in **Supplementary Appendix 4**.

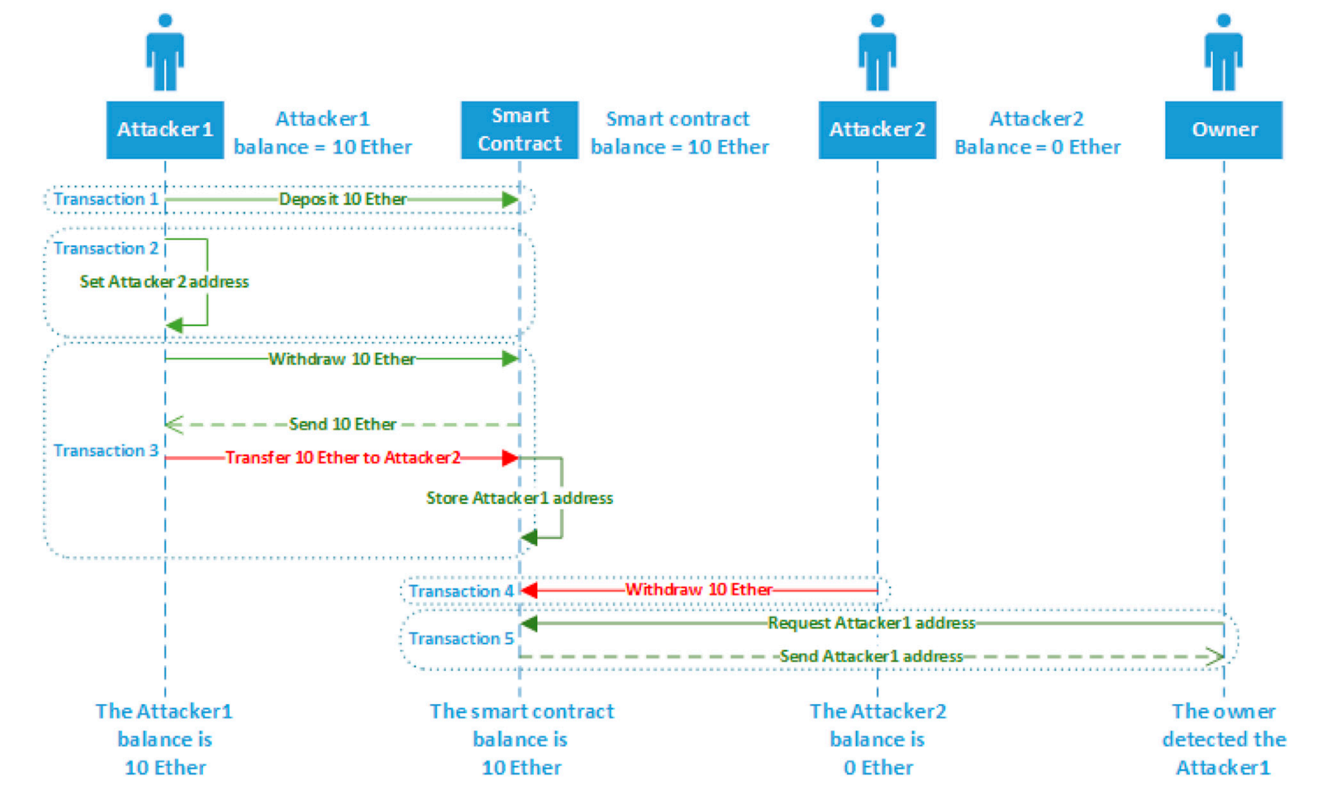

Attacker1
Attacker1 balance = 10 Ether
Smart Contract
Smart contract balance = 10 Ether
Attacker2
Attacker2 Balance = 0 Ether
Owner
Transaction 1
Deposit 10 Ether
Transaction 2
Set Attacker2 address
Withdraw 10 Ether
Send 10 Ether
Transaction 3
Transfer 10 Ether to Attacker2
Store Attacker1 address
Transaction 4
Withdraw 10 Ether
Request Attacker1 address
Transaction 5
Send Attacker1 address
The Attacker1 balance is 10 Ether
The smart contract balance is 10 Ether
The Attacker2 balance is 0 Ether
The owner detected the Attacker1

FIGURE 12 | UML sequence diagram for the second scenario in the second case study.

The scenario steps were as follows:

1. Deploying the *Bank{}* contract with funds equal to 10 Ethers.
2. Deploying the *Attacker1{}* contract by passing the *Bank{}* contract address as a parameter in the *Attacker1{}* constructor and with funds equal to 10 Ethers.
3. Deploying the *Attacker2{}* contract by passing the *Bank{}* contract address as a parameter in the *Attacker2{}* constructor.
4. Calling the *getAttackerBalance()* function in the *Attacker2{}* contract to check the balance, which is equal to 0 Ethers.
5. Depositing 10 Ethers from the *Attacker1{}* contract to the *Bank{}* contract.
6. Calling the *getAttackerBalance()* function in the *Attacker1{}* contract to check that the balance is equal to 0 Ethers.
7. Calling the *getBankLiquidity()* function in the *Bank{}* contract to check that the balance is equal to 20 Ethers.
8. Calling the *setAttacker2()* function in the *Attacker1{}* contract and passing the *Attacker2{}* contract address as a parameter.
9. Conducting the cross-function reentrancy attack by calling the *withdraw()* function in the *Attacker1{}* contract.
10. Calling *withdraw()* function in the *Attacker2{}* contract to steal 10 Ethers, which will fail because there are no funds for *Attacker2{}* in the *Bank{}* contract.
11. Calling the *getBankLiquidity()* function in the *Bank{}* contract to check the balance, which is equal to 10 Ethers.
12. Calling the *getAttackerBalance()* function in the *Attacker1{}* contract to check the balance, which is equal to 10 Ethers.
13. Calling the *getAttackerBalance()* function in the *Attacker2{}* contract to check the balance, which is equal to 0 Ethers, indicating that the attack is failed.
14. Calling the *getAttackerAddress()* in the *Bank{}* contract from the owner account to retrieve the attacker address.

In the second scenario, the cross-function reentrancy attack failed in leveraging the reentrancy vulnerability, and the *Attacker1{}* address is stored in the *Bank{}* contract.

The two scenarios in the two case studies followed the same attack steps and the solution, which is based on monitoring the difference between the contract balance and total of all participants' balances, proved that it can prevent the reentrancy attack during the execution time even though the reentrancy vulnerability existed in the contract. Additionally, the solution detected the attacker by storing its account address, which is accessible only by the targeted contract owner.

Discussion

From the previous two experiments, the proposed solution can produce the following positive results:

- Test case 1: single-function reentrancy attack, the proposed solution is successful in detecting the attacker and preventing the attack.
- Test case 2: cross-function reentrancy attack, the proposed solution is successful in detecting the attacker and preventing the attack

The solution's hypothesis relies on the fact that the difference between the two values, the contract balance and the total of all participants' balances, must be the same before and after any operation that changes the state of any contract. The solution can be implemented in different layers, as shown in the solution architecture, and can be implemented in different programming languages. Unlike the current defensive methodologies for reentrancy vulnerability, the proposed solution can work during the execution time after the smart contract deployment and can detect the attacker by recording its account address. The solution can be implemented in different ways; however, the main concept of the solution must be the same, which is based on monitoring the two values. The proposed solution can provide a secure and reentrancy-free Ethereum network. This solution targets the reentrancy vulnerability, which is one of the vulnerabilities in the Solidity layer.

CONCLUSION AND FUTURE RESEARCH

Our solution is based on the fact that the difference between the two values, the contract balance and the total balance of all participants, must be the same before and after any operation that changes the state of a contract. The solution can be implemented using different approaches and in different programming languages. Unlike the current defensive methods against reentrancy attacks, this solution works during execution, that is, after the smart contract is deployed, and can identify an attacker.

This article has analyzed the root cause of Ethereum reentrancy attacks and has proposed a solution with a proof-of-concept implementation, which can detect, prevent, and identify the account address of an attacker during the execution of a smart contract.

The result of this research prompts several possible future studies in enhancing the security of Ethereum. Firstly, the solution relies on the smart contract developers, a protocol layer solution can be established to protect all the smart contracts on the Ethereum network. The miners can check the transactions using this solution and if the transaction is considered malicious based on the solution, the miners can reject the transaction. Secondly, we invite the Ethereum community research validating the solution in the other approaches suggested here. This would evaluate the solution in different environments and measure the efficiency of these approaches to find what the best approach is in terms of security and efficiency. Finally, we shall extend this solution to detect and prevent other known smart contract vulnerabilities during the execution time through self-protection techniques carried out by the smart contract in case of attack. Since the smart contract is immutable against modification after the deployment, this research direction is very important.

There are many challenges to improving the security of smart contracts. A simple code flaw can have catastrophic results for a smart contract holding huge funds. Therefore, the industry and academia need to invest in future research to help create a more trustworthy Ethereum blockchain and improve the security of smart contracts.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/**Supplementary Material**; further inquiries can be directed to the corresponding author.

AUTHOR CONTRIBUTIONS

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2021.598780/full#supplementary-material.

REFERENCES

Adrian, O. R. (2018). "The blockchain, today and tomorrow," in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Timisoara, Romania, September 20–23, 2018 (IEEE), 458–462.

Alkhalifah, A., Ng, A., Chowdhury, M. J. M., Kayes, A. S. M., and Watters, P. A. (2019). "An empirical analysis of blockchain cybersecurity incidents," in 2019 IEEE Asia-Pacific conference on computer science and data engineering (CSDE), Melbourne, Australia, December 9–11, 2019 (IEEE), 1–8. doi:10.1109/CSDE48274.2019.9162381

Alkhalifah, A., Ng, A., Kayes, A. S. M., Chowdhury, J., Alazab, M., and Watters, P. A. (2020). "A taxonomy of blockchain threats and vulnerabilities," in *Blockchain for cybersecurity and privacy: architectures, challenges and applications*. (Boca Raton, FL: CRC Press Taylor & Francis), Chap. 1, 1–27.

Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2020). A survey on Ethereum systems security. *ACM Comput. Surv.* 53 (3), 1–43. doi:10.1145/3391195

Coblenz, M. (2017). "Obsidian: a safer blockchain programming language," in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Buenos Aires, Argentina, May 20–28, 2017 (IEEE), 97–99.

Dika, A., and Nowostawski, M. (2018). "Security vulnerabilities in Ethereum smart contracts,"in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Halifax, NS, July 30–August 3, 2018 (IEEE), 955–962.

Hung, C., Chen, K., and Liao, C. (2019). "Modularizing cross-cutting concerns with aspect-oriented extensions for solidity," in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Newark, CA, April 4–9, 2019 (IEEE), 176–181.

Lee, J. (2018). Patch transporter: incentivized, decentralized software patch system for WSN and IoT environments. *Sensors* 18 (2), 574. doi:10.3390/s18020574

Liu, J., and Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access.* 7, 77894–77904. doi:10.1109/access.2019.2921624

Luu, L., Chu, D. H., Olickel, H., Saxena, P., and Hobor, A. (2016). "Making smart contracts smarter," in ACM SIGSAC conference on computer and communications security, Vienna, Austria, October 25–27, 2016, (ACM), 254–269.

Madnick, S. (2020). Blockchain isn't as unbreakable as you think. *MIT Sloan Manag. Rev.* 61 (2), 65–70. doi:10.2139/ssrn.3542542

Prechtel, D., Gros, T., and Muller, T. (2019). "Evaluating spread of "gasless send" in Ethereum smart contracts," in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Canary Islands, Spain, June 24–26, 2019 (IEEE), 1–6.

Samreen, N. F., and Alalfi, M. H. (2020). "Reentrancy vulnerability identification in Ethereum smart contracts," The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, London, ON, February 18, 2020 (IEEE), 22–29.

Schrans, F., Eisenbach, S., and Drossopoulou, S. (2018). "Writing safe smart contracts in flint," in Conference companion of the 2nd international conference on art, science, and engineering of programming, New York, NY, April 9, 2018 ACM, 218–219.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). "SmartCheck: static analysis of Ethereum smart contracts," The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Gothenburg, Sweden, May 27–June 3, 2018 (IEEE), 9–16.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. (2018). "Securify: practical security analysis of smart contracts," in 2018 ACM SIGSAC conference on computer and communications security, Toronto, ON, October 15–19, 2018 (ACM), 67–82.

Zhang, W., Banescu, S., Pasos, L., Stewart, S., and Ganesh, V. (2019). "MPro: combining static and symbolic analysis for scalable testing of smart contract," in The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Berlin, Germany, October 28–31, 2019 (IEEE), 456–462.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.