2023 4th International Conference on Computation, Automation and Knowledge Management (ICCAKM) | 979-8-3503-9324-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAKM58659.2023.10449645

Preventing Ethereum Blockchain Re-entrancy Attacks using Smart Mutex Lock Sum

Nirbeh Kaur
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
nirbehkaur@gmail.com

Kapil Bhardwaj
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
kapil.e13779@cumail.in

Hemant Kumar Saini
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
hemantrhce@rediffmail.com

Shruti Kumari
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
shrutisinghcu9261@gmail.com

Shubham Kumar
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
shubhamprem888@gmail.com

Parneet Kaur
Department of CSE,UIE
Chandigarh University
Gharuan,Punjab,India
parneetkaur2002@gmail.com

***Abstract*—Since the smart contracts are being introduced, they are being subjected to various vulnerabilities. These vulnerabilities pose a major threat to participant's money and cause conflicts. Smart contracts are self executing contracts that use blockchain technology to transfer values among peers in which there is no control of a central authority to run and execute transactions. Reentrancy, one of the most important vulnerabilities in smart contracts, has caused millions of dollars in financial loss. As technology advances, many reentrancy solutions have been implemented and various detection approaches have been proposed. In this paper, it proposed a "Smart MutexLock" mechanism to prevent Ethereum reentrancy. Since the smart contracts on blockchain platforms like Ethereum do not directly support mutex locks due to their deterministic and asynchronous nature, it have used mutex locks in an effective and secure manner. The mutex mapping is used to keep track of whether a particular address is currently in a transaction to prevent reentrancy. The mutex variable is set to true before any external call, preventing further calls until the transaction is completed. Once the transaction is complete, the mutex variable is set back to false and user is notified to ensure a safe transaction**

Keywords—Ethereum, Smart contracts,Reentracy, vulnerability, Blockchain

I. Introduction

Blockchain is a decentralised digital ledger system that enables many users to manage a common database without depending on a single entity [1]. Cryptocurreny is a well known application of Blockchain. There are many crypto currencies like Bitcoin, Ethereum, Litecoin, Binance coin, Tether, Polkadot etc. that are in use in today's world , but out of all these Bitcoin and Ethereum are amongst the top. The main functions of bitcoin are as a store of value and a virtual currency. Ether can be used as a store of wealth and virtual currency. However, the decentralized Ethereum network also enables the development and operation of applications, smart contracts, and other network-based transactions. [2] This is the reason that Ethereum is one of the famous cryptocurrency that works on blockchain. Blockchain technology offers a numerable of transformative capabilities, including enhanced security, immutability, and decentralized data management. However, among its most pivotal innovations is the introduction of smart contracts. These self-executing agreements are like digital contracts which implement the set of rules without any third party intervention and encode contractual obligations in code and automatically enforce terms without the need for intermediaries, fundamentally reshaping traditional business processes. Smart contracts have emerged as a cornerstone of blockchain applications, fostering increased transparency and efficiency across various industries.

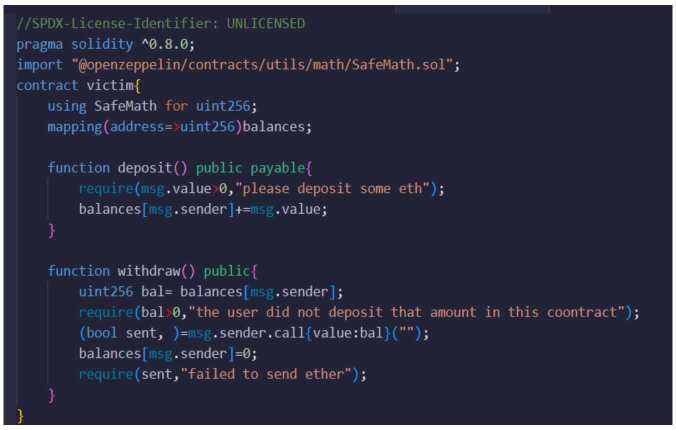
Since the smart contracts are self executing agreements, a small vulnerability in it can lead to a big loss of millions of money [3]. There are many incidents from the past that has caused a major loss to people's money. One such attack happened in 2016, which is known as "The DAO attack" which resulted in a loss of more than 3,600,000 million Ethers [4]. The famous DAO attack happened due to the reentrancy vulnerability in ethereum smart contracts which resulted in a hard fork, and eventually this hard fork lead to two versions of Ethereum blockchain - Ethereum (ETH) and Ethereum Classic (ETC), with the former rolling back the chain to recover the funds and the latter maintaining the original chain with no rollback [4]. Other than reentrancy attacks, there are other form of vulnerabilities also like Front-running, integer underflow and overflow, 51% attacks, logic error, incorrectly handled exceptions, block gas limit vulnerability etc. Although there is 'n' number of smart contract vulnerabilities, the reentrancy attacks are the one which has caused security threat to the Ethereum users.

Smart contracts are vulnerable to attacks due to various factors inherent to their design and implementation. The codes being developed by the developers contain flaws and errors which result in vulnerability as smart contracts are new platform [5]. Also, they are immutable which means that if the smart contracts are deployed once on the blockchain, they become immutable and cannot be altered [6]. Therefore, any vulnerabilities or errors in the code cannot be easily rectified without a significant impact on the network. [7]. As, the value of Ether is extremely high, the attackers are in the race of getting rich by exploiting a single vulnerability on smart contract [7].

The article is organized as follows. In section I smart contracts and its vulnerabilities are explained. Next exploring the problem and their current countermeasures in sectional. Section III of the article includes the *research questions* where it analyzed the need for research in this topic. In section IV *Methodology and proposed mechanism*, implemented the solution with its steps and flowchart. Then run code setup and analyzed its performance in section V. whose performance captured in with existing solutions in the section VI. Finally concluded with limitation for the future work in section VII.

II. Background

Reentrancy attacks occurs in smart contract when an attacker writes a malicious smart contract which repeatedly calls the withdraw function of victim's contract by forcing it to execute additional code by utilizing a fallback function to call back itself, without updating the balance, which result in transferring of entire money to attacker's account [8][9]. It is one of the most devastating attacks in Solidity smart contract which can cause loss of millions if suitable preventive measures are not taken properly [10].One of the most recent examples is the reentrancy vulnerability that was found in Ethereum's Constantinople hard fork. The Constantinople hard fork was a planned upgrade to the Ethereum network that was implemented on February 28, 2023 1. The hard fork was initially scheduled to occur on January 16, 2023, but it was delayed due to the discovery of reentrancy vulnerability. [11]. The Cream Finance attack [12] in which attacker was able to borrow and repay a single loan multiple times, leading to theft of $30 million cryptocurreny assets. Similar is the case of Lendf.me Protocol where $350000 of crypto assets was stolen [13] .The reentrancy vulnerable code of victim contract is given below:

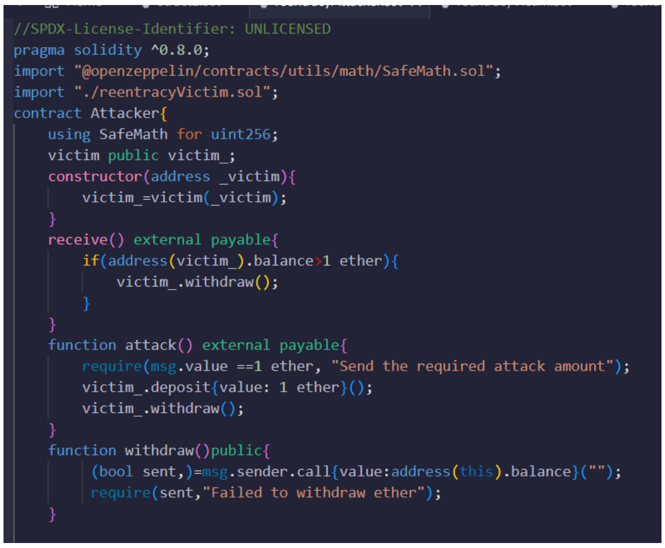


```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
contract victim{
    using SafeMath for uint256;
    mapping(address=>uint256)balances;

    function deposit() public payable{
        require(msg.value>0,"please deposit some eth");
        balances[msg.sender]+=msg.value;
    }

    function withdraw() public{
        uint256 bal= balances[msg.sender];
        require(bal>0,"the user did not deposit that amount in this coontract");
        (bool sent, )=msg.sender.call{value:bal}("");
        balances[msg.sender]=0;
        require(sent,"failed to send ether");
    }
}
```

Fig. 1. Re-entrancy vulnerable contract

In fig 1, the 'victim' contract is vulnerable to reentrancy attacks due to the way the 'withdraw' function is implemented. The contract lacks a mechanism to prevent multiple recursive calls to the 'withdraw' function within the same transaction. This absence of a mutex lock allows an attacker to exploit the function's reentrancy vulnerability, enabling them to repeatedly call the 'withdraw' function before the state changes are finalized.

The attacker will write a malicious smart contract in the following way:



```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "./reentracyVictim.sol";
contract Attacker{
    using SafeMath for uint256;
    victim public victim_;
    constructor(address _victim){
        victim_=victim(_victim);
    }
    receive() external payable{
        if(address(victim_).balance>1 ether){
            victim_.withdraw();
        }
    }
    function attack() external payable{
        require(msg.value ==1 ether, "Send the required attack amount");
        victim_.deposit{value: 1 ether}();
        victim_.withdraw();
    }
    function withdraw()public{
         (bool sent,)=msg.sender.call{value:address(this).balance}("");
         require(sent,"Failed to withdraw ether");
    }
```

Fig. 2. Malicious Attacker Contract

In fig 2, the 'Attacker' contract is created in a manner such that the reentrancy vulnerability in the 'victim' contract can be exploited. By initiating a series of recursive calls between the 'deposit' and 'withdraw' functions of the 'victim' contract, the attacker aims to drain the funds from the 'victim' contract and potentially disrupt its intended operations.

There have been many solutions that have been proposed to prevent reentrancy attacks on Ethereum. Several vulnerability detection tools have been made which provides security during the development of smart contracts [14]. Certain IDEs have been developed like Remix, Hardhat, Truffle, VScode, EthFiddle, IntelliJ IDEA etc which detects errors and vulnerabilities when the smart contracts are being deployed. Various high level programming languages have been introduced which provides additional functionalities for detecting and preventing vulnerabilities, errors and exceptions [15]. Other suggestions include some that are special to Solidity and could be instructive for others creating smart contracts in other languages, including utilizing modifiers solely for assertions [16].

Different researches have also been proposed to detect and prevent Ethereum smart contract vulnerability attacks.[23] The framework [17] analyzes smart contracts statically to identify potentially vulnerable functions and then uses dynamic analysis to precisely confirm Reentrancy vulnerability, thus achieving increased performance and reduced false positives. Another mechanism was proposed in which difference in amount before and after the transaction was equated to prevent reentrancy [18]. A hierarchical CPN modeling method has been proposed by researchers to analyze potential security vulnerabilities at the contract's source code level [19]. Other researchers proposed a deep learning based two-phase smart contract debugger for reentrancy vulnerability, named as ReVulDL: Reentrancy Vulnerability Detection and Localization [20].

III. Research Questions

There are several questions that came to the minds that after so many existing solutions and proposed solutions by different researchers, what makes it possible for the attackers to exploit vulnerabilities and steal money. The following questions have been proposed:

Q1. What are the limitations of the current existing solutions?

Q2. How to implement a more secure smart contract free from reentrancy attacks?

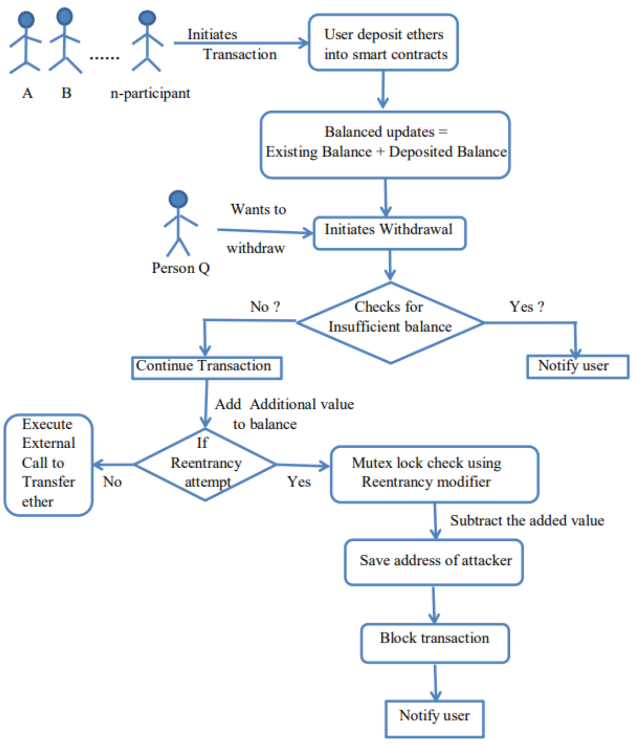
The answer of first question is that the mechanisms that have been proposed works on some predefined patterns, if the attacker uses new pattern to steal money, then these solutions will fail gradually [21]. The solutions that are existing already are only applicable if they are used before the deployment of smart contracts [21]. Reentrancy attacks typically occur at the application layer in the context of smart contracts. Tools that operate at the application layer primarily focus on analyzing the smart contract code and identifying potential vulnerabilities within the codebase itself.

The second question should be answered only by implementing a more secure and feasible mechanism which include smart contract code analysis, vulnerability detection at application layer, blockchain layer and also at consensus layer.

IV. Methodology and Proposed Mechanism

In the solution that it has proposed, it used the concept of mutex locks, which are also known as mutual exclusion locks; Mutex locks are a synchronization technique to manage access to shared resources. The advantage of using Mutex locks is that it ensures that only one instance of a function can be executed at a time in the smart contracts, hence preventing reentrancy attacks. Since, the proposed solution revolves around the concept of mutex locks, it have named this mechanism as "MutexLockSum Mechanism". "The MutexLockSum contract" is a secure smart contract which can be deployed on the Ethereum blockchain... It is equipped with advanced security measures, including mutex locks, to prevent reentrancy attacks and ensure the integrity of transactions. Users can interact with the contract by depositing and withdrawing ether securely, and perform sum operations on specified input values. The contract maintains a mapping of user balances and employs mutex locks to control access and prevent multiple recursive calls, thereby ensuring the sequential and secure execution of critical functions. Furthermore, the contract provides a notification mechanism for users and emits various events for logging and monitoring purposes. Overall, the MutexLockSum contract prioritizes security and reliability, demonstrating best practices in smart contract development and ensuring a robust and trustworthy transaction environment."

Since the smart contracts on blockchain platforms like Ethereum do not directly support mutex locks due to their deterministic and asynchronous nature, it have used mutex locks in an effective and secure manner. The mutex mapping is used to keep track of whether a particular address is currently in a transaction to prevent reentrancy. The mutex variable is set to true before any external call, preventing further calls until the transaction is completed. Once the transaction is complete, the mutex variable is set back to false.The flowchart of the mechanism is given:



A
B
......
n-participant
Initiates Transaction
User deposit ethers into smart contracts
Balanced updates = Existing Balance + Deposited Balance
Person Q
Wants to withdraw
Initiates Withdrawal
Checks for Insufficient balance
No ?
Yes ?
Notify user
Continue Transaction
Add Additional value to balance
If Reentrancy attempt
No
Yes
Execute External Call to Transfer ether
Mutex lock check using Reentrancy modifier
Subtract the added value
Save address of attacker
Block transaction
Notify user

Fig. 3. Flowchart of proposed solution

In fig 3, the flow chart is implemented in the following steps:

1. Begin MutexLockSum contract

2. Define the using statement for SafeMath library

3. Define private mapping 'balances' to store the balances of users

4. Define private mapping 'mutex' to handle the mutex locks for users

5. Define various events for logging purposes: SumCompleted, MutexLocked, MutexUnlocked, AttackAttempted, and Notification

6. Implement the 'noReentrancy' modifier to prevent reentrancy attacks

a. Check if the mutex lock for the current user is inactive

b. Activate the mutex lock for the current user

c. Emit the MutexLocked event

d. Execute the function logic

e. Release the mutex lock for the current user

f. Emit the MutexUnlocked event

7. Implement the 'deposit' function to allow users to deposit ether into the contract

a. Check if the deposited value is greater than 0

b. Update the user's balance with the deposited amount

8. Implement the 'withdraw' function to allow users to withdraw their funds with the no Reentrancy modifier

a. Check if the user has sufficient balance for withdrawal

b. Store the amount to be withdrawn

c. Reset the user's balance to 0

d. Attempt to transfer the specified amount to the user

e. If the transfer fails, emit the Attack Attempted event and notify the user about the attempted attack

9. Implement the 'sum' function to perform the sum operation with the no Reentrancy modifier

a. Check if the user has sufficient balance for the sum operation

b. Perform the sum operation for the provided input values

c. Update the user's balance by subtracting the amounts used in the sum operation

d. Emit the Sum Completed event with the computed result

e. Return the result of the sum operation

10. Implement the 'get Balance' function to retrieve the balance of the user

11. Implement the 'notify User' function to emit a notification event for the user with a specific message

12. Implement the '_safe Transfer' internal function to safely transfer ether to a specified address

a. Attempt to transfer the specified amount to the provided address

b. Return the success status of the transfer

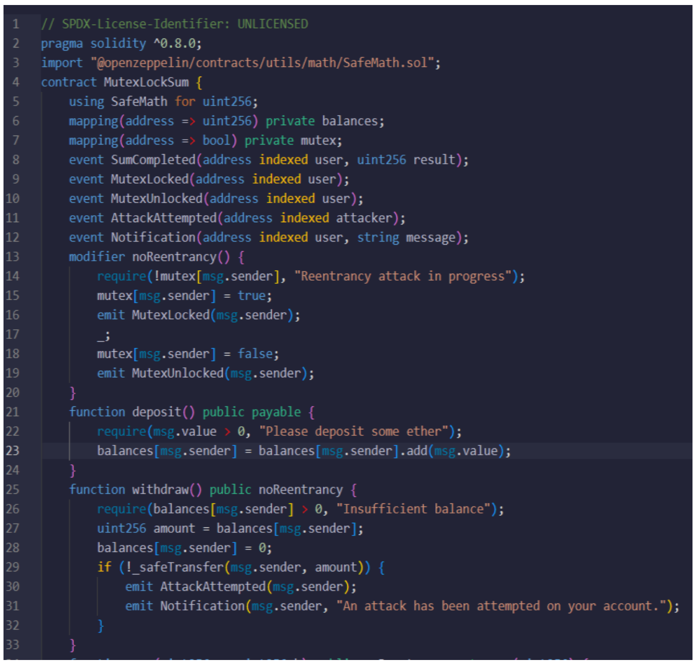
13. End MutexLockSum contract

V. Experimentation

The Remix IDE has been used to program the MutexLockSum smart contract. This mechanism is implemented using Solidity programming language. The code is executed in the following steps:

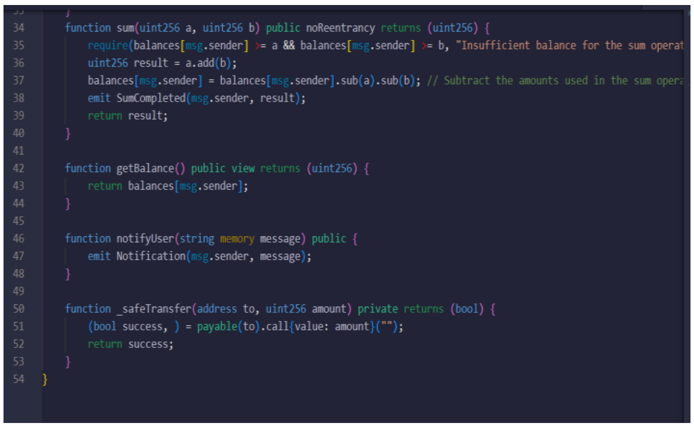
Mapping is done to store user balances as well as to handle mutex locks. The ***event SumCompleted*** will happen upon successful completion of the sum operation. The "***event MutexLocked***" will be emitted when the mutex lock is activated. Similar is the functionality of all other events like

Event MutexUnlocked, ***event Attack Attempted*** and ***event Notification***. The modifier ***no Reentrancy ()*** is used to prevent reentrancy attacks. In this modifier, the mutex locks will be activated. Then the different functions will be performing their desired functionality.



```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
contract MutexLockSum {
    using SafeMath for uint256;
    mapping(address => uint256) private balances;
    mapping(address => bool) private mutex;
    event SumCompleted(address indexed user, uint256 result);
    event MutexLocked(address indexed user);
    event MutexUnlocked(address indexed user);
    event AttackAttempted(address indexed attacker);
    event Notification(address indexed user, string message);
    modifier noReentrancy() {
        require(!mutex[msg.sender], "Reentrancy attack in progress");
        mutex[msg.sender] = true;
        emit MutexLocked(msg.sender);
        _;
        mutex[msg.sender] = false;
        emit MutexUnlocked(msg.sender);
    }
    function deposit() public payable {
        require(msg.value > 0, "Please deposit some ether");
        balances[msg.sender] = balances[msg.sender].add(msg.value);
    }
    function withdraw() public noReentrancy {
        require(balances[msg.sender] > 0, "Insufficient balance");
        uint256 amount = balances[msg.sender];
        balances[msg.sender] = 0;
        if (!_safeTransfer(msg.sender, amount)) {
            emit AttackAttempted(msg.sender);
            emit Notification(msg.sender, "An attack has been attempted on your account.");
        }
    }
```

(a)



```
    function sum(uint256 a, uint256 b) public noReentrancy returns (uint256) {
        require(balances[msg.sender] >= a && balances[msg.sender] >= b, "Insufficient balance for the sum operat
        uint256 result = a.add(b);
        balances[msg.sender] = balances[msg.sender].sub(a).sub(b); // Subtract the amounts used in the sum opera
        emit SumCompleted(msg.sender, result);
        return result;
    }

    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }

    function notifyUser(string memory message) public {
        emit Notification(msg.sender, message);
    }

    function _safeTransfer(address to, uint256 amount) private returns (bool) {
        (bool success, ) = payable(to).call{value: amount}("");
        return success;
    }
}
```

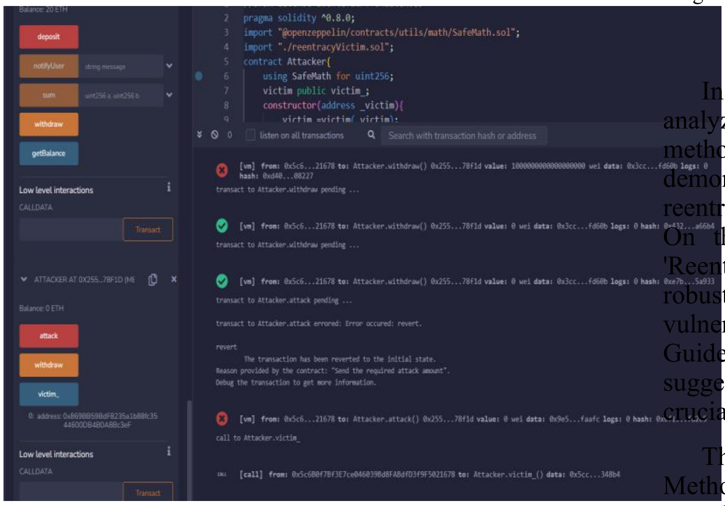
(b)

Fig. 4. (a) and (b) exploring function code for MutexLockSum

In fig 4, The MutexLockSum contract is built with a focus on preventing reentrancy attacks by using mutex locks. The 'MutexLockSum' contract presented here is a robust implementation that prioritizes security by utilizing mutex locks to prevent reentrancy attacks. The contract maintains private mappings for user balances and mutex locks, allowing it to securely manage user funds and control access to critical functions. Several events are defined within the contract to facilitate comprehensive event logging and monitoring, including 'SumCompleted', 'MutexLocked', 'MutexUnlocked', 'AttackAttempted', and 'Notification'. The contract's 'noReentrancy' modifier serves as a crucial security measure, ensuring that no reentrant calls can be made within the same transaction. Users can deposit Ether into the contract securely using the 'deposit' function, which effectively adds the deposited value to the sender's balance. Similarly, the 'withdraw' function enables users to withdraw their funds securely, performing thorough checks for sufficient balances and executing a safe transfer of the corresponding Ether amount back to the user. Moreover, the 'sum' function allows users to perform sum operations on specified input values, ensuring that sufficient user balances are available before processing the transaction. The contract meticulously updates the user's balance to reflect the outcome of the sum operation, further ensuring the integrity and accuracy of the transaction. Additionally, users can trigger custom notifications with specific messages using the 'notifyUser' function, enhancing the contract's communication capabilities. Internally, the '_safeTransfer' function guarantees secure Ether transfers to designated addresses, promoting the overall resilience and reliability of the contract's fund management system. By incorporating these key security features and emphasizing strict checks and secure transaction protocols, the 'MutexLockSum' contract exemplifies best practices for mitigating reentrancy attacks and fostering a secure transaction environment within the Ethereum blockchain.

VI. Experimental Results

After analysis and running the test code on remix idea itself, the MutexLockSum Mechanism has provided secure and feasible way to prevent reentrancy attacks. The malicious attacker contract in the part II. Of this paper failed to steal money if the proposed mechanism is applied to set up mutex locks. In fig 5, If it call the withdraw function of attacker using an ethereum account, an error is being shown depicting invalid transaction.



```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "./reentracyVictim.sol";
contract Attacker{
    using SafeMath for uint256;
    victim public victim_;
    constructor(address _victim){
        victim =victim( victim);
```

```
[vm] from: 0x5c6...21678 to: Attacker.withdraw() 0x255...78f1d value: 1000000000000000000 wei data: 0x3cc...fd60b logs: 0
hash: 0xd40...08227
transact to Attacker.withdraw pending ...

[vm] from: 0x5c6...21678 to: Attacker.withdraw() 0x255...78f1d value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0x432...a66b4
transact to Attacker.withdraw pending ...

[vm] from: 0x5c6...21678 to: Attacker.withdraw() 0x255...78f1d value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0xe7b...5a933
transact to Attacker.attack pending ...

transact to Attacker.attack errored: Error occured: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "Send the required attack amount".
Debug the transaction to get more information.

[vm] from: 0x5c6...21678 to: Attacker.attack() 0x255...78f1d value: 0 wei data: 0x9e5...faafc logs: 0 hash: 0x[illegible]
call to Attacker.victim_

[call] from: 0x5c6B0f7BF3E7ce046039Bd8FABdfD3f9F5021678 to: Attacker.victim_() data: 0x5cc...348b4
```

Fig. 5. Failure of re-entrancy attack

It have compared the effectiveness of the solution i.e. MutexLockSum with the existing solutions like Time delay method, Checks- Effects interaction pattern, Reentrancy guard method and best practice guidelines and it have found that the proposed solution provides high effectiveness in preventing reentrancy. The comparison of the proposed mechanism with exixting mechanisms is given below:

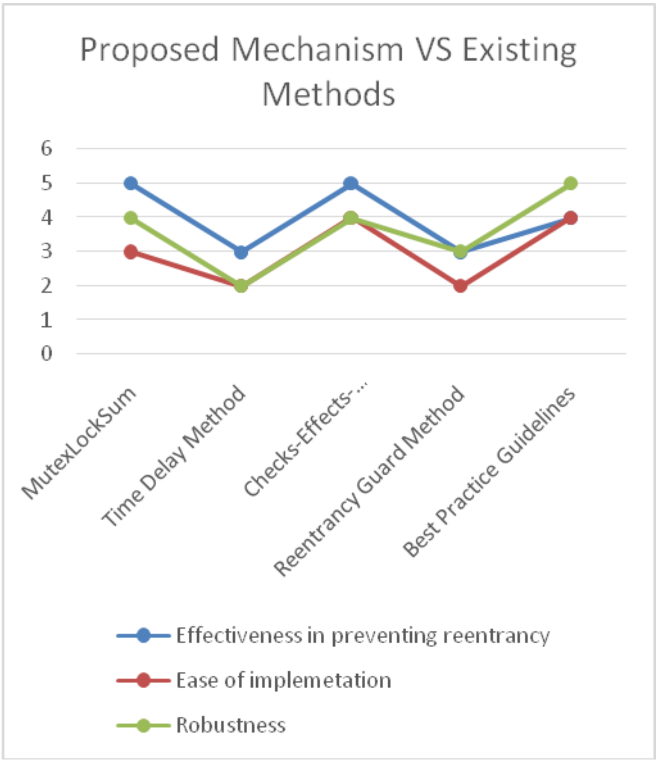


Fig. 6. Graph showing the comparison of proposed vs existing methods

In fig 6, a comparison graph is made to examine and analyze the experimental results. The 'MutexLockSum' method and the 'Checks-Effects-Interactions Pattern' demonstrate high effectiveness and robustness in preventing reentrancy attacks, as they both scored well in these aspects. On the other hand, the 'Time Delay Method' and the 'Reentrancy Guard Method' show moderate effectiveness and robustness, indicating that they might have some vulnerabilities to sophisticated attacks. The 'Best Practice Guidelines' appear to be the most robust and effective, suggesting that adherence to general security standards is crucial in smart contract development.

The 'Time Delay Method' and the 'Reentrancy Guard Method' appear to be the easiest to implement, requiring less complexity in their integration. In contrast, the 'MutexLockSum' method and the 'Checks-Effects-Interactions Pattern' show moderate to complex implementation, implying that they might demand more intricate coding and design considerations [22]. The 'Best Practice Guidelines' also require a moderate to complex implementation, emphasizing the importance of thorough security protocols in the development process.

Therefore, the proposed mechanism enlightens more on preventing reentrancy, although its implementation is bit complex as compared to others. So, there is a proper need to analyze and advanced work on this mechanism so that it can be implemented at the developmental level.

VII. Experimental Results

The proposed paper has analyzed the underlying causes of Ethereum reentrancy attacks and has presented a smart solution that can identify, stop, and detect an attacker's account address while a smart contract is being executed.

The findings of this study suggest a number of potential directions for more research in order to strengthen Ethereum security. In order to safeguard all of the smart contracts on the Ethereum network, a protocol layer solution might be built, which first depends on the developers of smart contracts. The solution allows the miners to check the transactions, and if the solution determines that a transaction is malicious, the miners can reject it. This research area is crucial because smart contracts cannot be changed once they have been deployed [22].

The improvement of smart contracts' security faces numerous obstacles. For a smart contract holding significant sums of money, a simple code error can have disastrous effects. As a result, future research must be funded by both the private sector and the academic community in order to strengthen the security and integrity of smart contracts and build a more reliable Ethereum blockchain.

References

[1] "McKinsey Technology Trends Outlook 2022," August 24, 2022

[2] 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

[3] A. Alkhalifah, A. Ng, M. J. M. Chowdhury, A. S. M. Kayes and P. A. Watters, "An Empirical Analysis of Blockchain Cybersecurity Incidents," 2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), Melbourne, VIC, Australia, 2019, pp. 1-8, doi: 10.1109/CSDE48274.2019.9162381..

[4] D. Vangulick, B. Cornelusse and D. Ernst, "Blockchain: A Novel Approach for the Consensus Algorithm Using Condorcet Voting Procedure," in 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), Newark, CA, USA, 2019 pp. 1-10. doi: 10.1109/DAPPCON.2019.00011.

[5] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming (Programming '18). Association for Computing Machinery, New York, NY, USA, 218–219. https://doi.org/10.1145/3191697.3213790

[6] Madnick, Stuart E., Blockchain Isn't as Unbreakable as You Think (November 1, 2019). Available at SSRN: https://ssrn.com/abstract=3542542 or http://dx.doi.org/10.2139/ssrn.3542542

[7] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. Springer-Verlag, Berlin, Heidelberg, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[8] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen and B. Roscoe, "ReGuard: Finding Reentrancy Bugs in Smart Contracts," 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 2018, pp. 65-68.

[9] N. Fatima Samreen and M. H. Alalfi, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), London, ON, Canada, 2020, pp. 22-29, doi: 10.1109/IWBOSE50093.2020.9050260.

[10] Z. Pan, T. Hu, C. Qian and B. Li, "ReDefender: A Tool for Detecting Reentrancy Vulnerabilities in Smart Contracts Effectively," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 2021, pp. 915-925, doi: 10.1109/QRS54544.2021.00101.

[11] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. ACM Comput. Surv. 53, 3, Article 67 (May 2021), 43 pages. https://doi.org/10.1145/3391195

[12] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. In Proceedings of the 45th International Conference on Software Engineering (ICSE '23). IEEE Press, 295–306. https://doi.org/10.1109/ICSE48619.2023.00036

[13] He Y, Dong H, Wu H, Duan Q. Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN. *Electronics*. 2023; 12(10):2152. https://doi.org/10.3390/electronics12102152.

[14] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2023. Reentrancy Vulnerability Detection and Localization: A Deep Learning Based Two-phase Approach. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 83, 1–13. https://doi.org/10.1145/3551349.3560428

[15] F. Sparbrodt and M. García-Valls, "Digesting smart contracts in Ethereum blockchain networks," 2022 5th Conference on Cloud and Internet of Things (CIoT), Marrakech, Morocco, 2022, pp. 60-66, doi: 10.1109/CIoT53061.2022.9766685.

[16] He Y, Dong H, Wu H, Duan Q. Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN. *Electronics*. 2023; 12(10):2152. https://doi.org/10.3390/electronics12102152

[17] T. A. Usman, A. A. Selçuk and S. Özarslan, "An Analysis of Ethereum Smart Contract Vulnerabilities," 2021 International Conference on Information Security and Cryptology (ISCTURKEY), Ankara, Turkey, 2021, pp. 99-104, doi: 10.1109/ISCTURKEY53027.2021.9654305.

[18] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. 2023. A survey on smart contract vulnerabilities: Data sources, detection and repair. Inf. Softw. Technol. 159, C (Jul 2023). https://doi.org/10.1016/j.infsof.2023.107221.

[19] Alexander Mense and Markus Flatscher. 2018. Security Vulnerabilities in Ethereum Smart Contracts. In Proceedings of the 20th International Conference on Information Integration and Web-based Applications &amp; Services (iiWAS2018). Association for Computing Machinery, New York, NY, USA, 375–380. https://doi.org/10.1145/3282373.3282419 .

[20] Noama Fatima Samreen and Manar H. Alalfi. 2020. A survey of security vulnerabilities in ethereum smart contracts. In Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON '20). IBM Corp., USA, 73–82..

[21] Dong, C., Li, Y., Tan, L. (2020). A New Approach to Prevent Reentrant Attack in Solidity Smart Contracts. In: Si, X., *et al.* Blockchain Technology and Application. CBCC 2019. Communications in Computer and Information Science, vol 1176. Springer, Singapore. https://doi.org/10.1007/978-981-15-3278-8_6

[22] Manoj, R. & Joshi, Sandeep (2023) Ensuring wallet application security by resolving reentrancy attacks in blockchain smart contracts, *Journal of Discrete Mathematical Sciences and Cryptography,* 26:3, 927-937, DOI: 10.47974/JDMSC-1779

[23] Sharma, V., Manocha, T., Garg, S., Sharma, S., Garg, A., & Sharma, R. (2023, February). Growth of Cyber-crimes in Society 4.0. In *2023 3rd International Conference on Innovative Practices in Technology and Management (ICIPTM)* (pp. 1-6). IEEE