Adaptation of PCC in Smart Contract

Submitted in partial fulfillment of the requirements
of the degree of

Masters of Technology

by

Hrishikesh Saloi
(Roll No. 213050057)

Supervisors:

Prof. Virendra Singh
Prof. R.K. Shyamasundar

Computer Science Engineering
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2023

Thesis Approval

This thesis entitled **Adaptation of PCC in Smart Contract** by **Hrishikesh Saloi** is approved for the degree of **Masters of Technology**.

Examiners:

.....

.....

.....

.....

Supervisor:

.....

Chairperson:

.....

Date: .....

Place: .....

Abstract

This master's thesis dissertation examines the vulnerabilities present in Solidity smart contracts that have emerged over the years and explores the available solutions to mitigate these risks. Additionally, the limitations of hooks in XRPL (XRP Ledger) are reviewed, and alternative methods are proposed to overcome these limitations. By capturing the invariants of XRPL hooks and presenting them as Proof-Carrying Code (PCC), this research establishes a framework to ensure the correctness of hooks, instilling confidence in their usage. Furthermore, the study demonstrates how the coroutine structure, widely employed in earlier practices, can effectively be utilized for specifying smart contracts.In order to illustrate the prevailing vulnerabilities in smart contracts, the research employs co-routines. Through this approach, it becomes evident that co-routines address certain vulnerabilities, such as reentrancy, in a natural manner, while others can be readily avoided with proper implementation.This dissertation contributes to the field of blockchain security by providing insights into the vulnerabilities of Solidity smart contracts, offering solutions to address them, and highlighting the potential of co-routines for improved contract specification. The findings of this research aim to enhance the robustness and security of smart contract development and deployment.

Contents

Chapter 1

Introduction

1.1 Blockchain

Blockchain is a distributed ledger technology that allows for secure, transparent, and tamper-proof recording of transactions. It is a system of recording information in a way that makes it difficult or impossible to change, hack, or cheat the system. The history of blockchain can be traced back to the early 1990s, when researchers Stuart Haber and W. Scott Stornetta developed a system for timestamping digital documents. Their system used a cryptographic hash function to create a unique digital fingerprint for each document, which was then stored in a linked list of blocks. This made it possible to verify the authenticity and integrity of the documents, as well as to prevent them from being tampered with. In 2008, an anonymous person or group of people under the pseudonym **Satoshi Nakamoto** published a white paperNakamoto (2008) that described a new type of digital currency called Bitcoin. Bitcoin used blockchain technology to record transactions, making it a secure and decentralized way to transfer value. The development of Bitcoin led to a surge of interest in blockchain technology, and many other applications have since been developed. These applications range from supply chain manage-

ment to voting to healthcare. There are many different versions of blockchains, each with its own unique features and applications. Some of the most popular blockchains include **Bitcoin**, **Ethereum**,**Hyperledger** Fabric. **Bitcoin** and **Ethereum** are the two most popular blockchains. They are both public blockchains, which means that anyone can participate in the network. However, there are some key differences between the two blockchains.Bitcoin is a peer-to-peer payment system. It is designed to be a decentralized way to transfer value. Ethereum is a platform for building dApps(decentralized apps). It is designed to be more versatile than Bitcoin and can be used for a variety of applications.

1.2 Smart Contract

Smart contracts are self-executing contracts turned programs with predefined conditions and terms encoded into code. In 2015, Ethereum introduced the concept of smart contracts to the blockchain ecosystem, allowing developers to create and deploy decentralized applications (dApps) through the use of smart contracts.The Ethereum Virtual Machine (EVM) is a runtime environment that executes smart contracts on the Ethereum network. It acts as a low-level stack machine with a minimal instruction set. They facilitate secure, transparent, and automated transactions without the need for intermediaries Despite the security features of the EVM, there have been a number of attacks on smart contracts. These attacks have exploited vulnerabilities in the code of smart contracts or in the way that smart contracts are used. Some of the most common attacks on smart contracts include: reentrancy vulnerability, parity wallet atack etc.

1.3 Solidity

Solidity is a high-level programming language used for developing smart contracts on the Ethereum blockchain. It enables developers to define data structures, functions, and events within contracts. However, Solidity is not immune to vulnerabilities. The reentrancy vulnerability allows an attacker to repeatedly call a contract before it completes, resulting in unauthorized fund transfers. The Parity wallet attack exploited a vulnerability in the multisig wallet contract,

leading to significant financial losses. Other vulnerabilities include integer overflow/underflow, unhandled exceptions, and malicious input handling. Solidity developers must be aware of these vulnerabilities and implement proper security measures, such as using the "pull" over "push" payment pattern and conducting thorough code audits, to ensure the integrity and security of their smart contracts.

Chapter 2

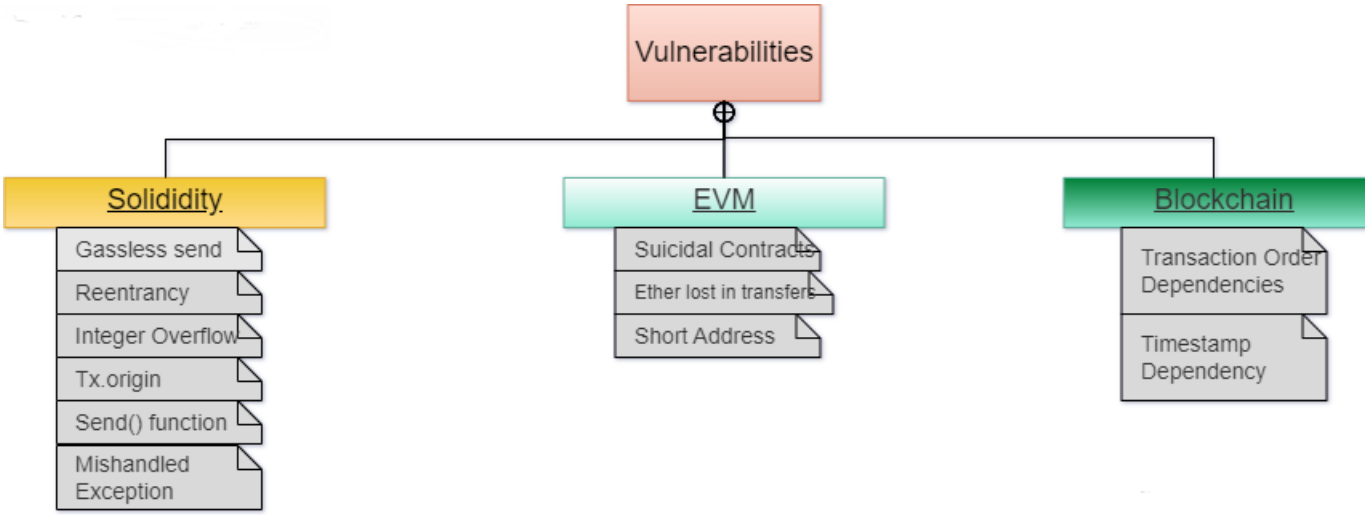
Smart Contracts and their vulnerabilties

Several programming languages are available for writing smart contracts, each tailored to specific blockchain platforms. Solidity is the most popular language for developing smart contracts on Ethereum, offering a JavaScript-like syntax and a wide range of features. Solidity is a high-level programming language used for developing smart contracts on the Ethereum blockchain. It enables developers to define data structures, functions, and events within contracts. However, Solidity is not immune to vulnerabilities. The reentrancy vulnerability allows an attacker to repeatedly call a contract before it completes, resulting in unauthorized fund transfers. The Parity wallet attack exploited a vulnerability in the multisig wallet contract, leading to significant financial losses. Other vulnerabilities include integer overflow/underflow, unhandled exceptions, and malicious input handling. Solidity developers must be aware of these vulnerabilities and implement proper security measures, such as using the "pull" over "push" payment pattern and conducting thorough code audits, to ensure the integrity and security of their smart contracts.

2.1 Issues and Challenges in the correctness of the Smart Contract

Smart Contract is a software application that runs on the Ethereum Blockchain. Its immutable nature stops any users to tamper with it once it is deployed in the Blockchain. However, it carries the capacity to change the global state of Ethereum. The global state can be changed by invoking the functions of the Smart Contract which updates the variable that lies in the memory section of the smart contract. Any undesired behavior of the smart Contract may lead to huge financial loss. The underlying reasons for the undesired behavior may vary. The Solidity language contains inbuilt functions which behave weirdly in some special conditions. For example, send() and transfer() are the functions used for transferring ether from one account to another. However, when the transaction fails transfer() raises an error but send() does not. This may create some serious problems. Again when there is a dependency between two transactions, there is no guarantee that the transactions will be executed in the order it was supposed to be. Ethereum Virtual Machine(EVM) is the actual stack based-machine where all the transactions and deployed contracts are executed. EVM allows the user to destroy a smart contract. However the contracts are not destroyed, only the mappings of the contract address with its code and memory section are removed. The address remains forever in the Blockchain becoming an orphan address (discussed in detail here**??**) with no owner. This orphan address also behaves in a very strange manner. This strange behavior of the smart contract brings in the necessity of a thorough study of the smart contract vulnerabilities. The reasons behind the strange behavior of the smart contract can be broadly categorized into 3 categories i.e., vulnerabilities that occur due to the semantics of solidity language, the execution of the transactions in the EVM, and the working of Blockchain itself. The vulnerabilities and their category is listed the fig . A detailed explanation of each of the vulnerabilities is discussed in the coming sections.

Figure 2.1: Category of Vulnerabilities



2.2 Classes of vulnerabilities

In the paper Nikolić et al. (2018) by Nikolić, Ivica, et al. , the authors divided vulnerable contracts into four types of vulnerabilities (1) Prodigal , (2) Suicidal , (3) Greedy and (4) Posthumous Contracts.

2.2.1 Prodigal Contracts

Users who have deposited money at a certain point in the contract can often withdraw that money. However, there are certain instances where money is sent to an arbitrary contract that has never dealt with a contract. Prodigal Contracts are the contracts that have these weaknesses.

Figure 2.2: A Bounty Contract

```
1   1 function payout ( address [] recipients ,
2   2 uint256 [] amounts ) {
3   3 require ( recipients . length == amounts . length ) ;
4   4 for ( uint i = 0; i < recipients . length ; i ++) {
5   5 /* ... */
6   6 recipients [ i ]. send ( amounts [ i ]) ;
7   7 }}
8
```

Take a look at the Bounty contract with the code fragment in Figure 3.1. This contract gathers Ether from various places and distributes bounties to a predetermined group of recipients. The payout function sends specific amounts of ether to a list of recipients. It is evident from the function's definition that the recipients and the amounts are specified by the inputs, and anyone can call the function (i.e., the function has no sender restrictions). As a result, any user can use this function to send all of the Ether in the contract to the addresses of her choosing. It is not necessary for the address to be one of the addresses that has dealt with the contract.

2.2.2 Suicidal Contracts

Several Contracts a *suicide()* function to self destroy the contract for various reason. But the ownership of who can invoke this function is not well defined. This type of privileged function should only be called be owners and none else. Owner not defining properly may lead to some serious consequences. The contract which can be self destructed by an arbitrary account falls under this category. The second parity wallet attack as discussed in section 2.2.5 can be seen as an example of such a contract where a user invoked the kill() function and wallets that use this contract were frozen forever.

2.2.3 Greedy Contracts

This category includes contracts that let users deposit money but don't have a way to send ethers out, locking ethers for all time. Many other multisigWallet-like contracts that held Ether in the Parity contract example discussed in section2.2.5 used functions from Library contracts. The wallet contracts are left without the ability to send ethers after the Library contract was killed. As a result, the ethers were trapped in the contracts for all time with no way out.A $200 million US worth of ether was locked indefinitely as a result of this vulnerability. Greedy contracts can also result from more obvious mistakes. The majority of these errors happen in contracts that accept Ether but either have no instructions at all for sending Ether elsewhere (such as bytecode instructions for send, call, and transfer), or these instructions are not reachable.

2.2.4 Posthumous

When a contract is killed, the Blockchain system completely clears out all of the global variables and code associated with the contract. No contracts' features are accessible after a kill. The problem is that the user can still call the dead contract's functions, even though doing so will be useless, and if any ether is sent along with the call, it is added to the balance of the contract address. The Posthumus Contract category includes contracts of this nature that have been terminated but are still being paid for by customers..A broader view of the killed contracts in given in next chapter

Over the years the many attacks took place in ethereum solidity smart contract. The famous ones like Reentrancy, parity are already studied and explored by many. The following sections includes some less famous but serious possible vulnerability and attacks.

2.2.5 Parity Multisig Attack

Blockchain transaction depends on public-private key pair for transactions to be made. Public key of the receiver is needed to send money while private key of the sender is needed in order to withdraw money from his account. Due to the immutable nature of the blockchain, if a private key is stolen or lost , user may loose all his fund.Thus if not properly stored, user may suffer heavy financial losses. In order to deal with this issue, an application called "WALLET" was devised whose primary duty was to store the private keys of the users and help user with transactions. But still using a single public-private key pair is vulnerable to attacks and thief being a single-point of error. In 2012, **BITGO** introduced a MULTI-SIG technology to tighten the security of Blockchain transactions where multiple signatures are required to make a transaction instead of one. Here , a single private key is converted into multiple signatures and are stored in different devices or online servers. The user need to use a subset of these keys to invoke a transaction. In 2017 , **THE PARITY TECHNOLOGY** team launched the MULTISIG-WALLET for Ethereum. This wallet was considered to be the safest until the attack occurred in the parity wallet. But using multi-sig technology with ethereum contracts were very expensive. It cost around 97% more gas compared to using a single key pair authentication. To solve this issue,

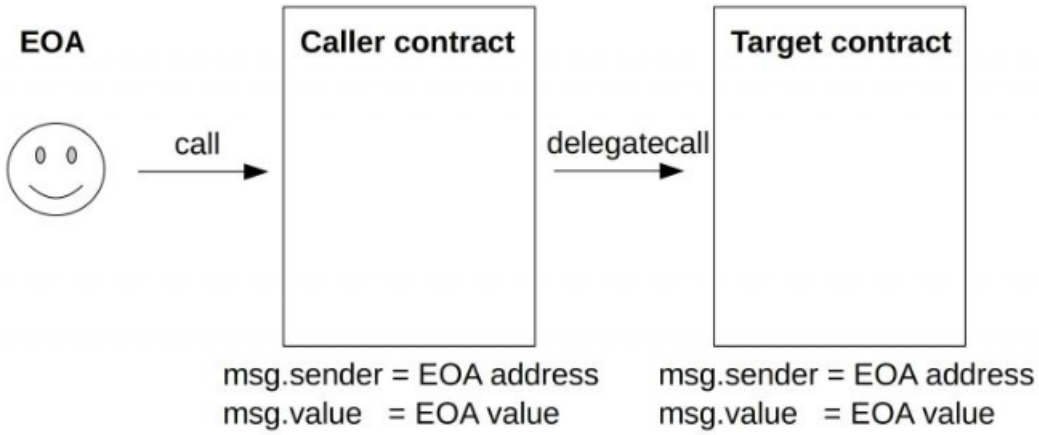
the parity team used shared libraries. The Parity Wallet is now divided into two contracts : (1) the *"WalletLibrary"* contract that contains the logic of multi-sig functionalities , (2) the *wallet* contract which contains the functions to be invoked by the users. Multiple *Wallet* contracts can refer to the *WalletLibrary* for the functionalities. With this gas consumption were cut down by 97%.

Before discussing the parity attacks, it is important to understand how the functions calls are made internally to the *WalletLibrary* by the *Wallet* contracts. There are 3 methods of calling a function in solidity (1) **CALL** (2) **CALLCODE** (3) **DELEGATECALL** . Parity used the **DELEGATECALL** option to invoke the functions of *"WalletLibrary"* contract

DelegateCall

Though syntactically DelegateCallTechnologies (2017) is same as **CALL** but it differs in behavior. One primary differences is the context.With **DelegateCall**, the target contract works under the same context as the caller contract. It means the target contract can have access to the caller contract state variables and sees the same global variable like msg.sender or msg.value.

Figure 2.3: Demonstration of working of Delegate Call



Example:

Let us consider two Contracts: (1)Caller Contract (2)Lib Contract; where CallerContract will make delegate calls to Lib Contract.The *CallerContract* has two state variables: (1)owner (2) lib . The variable *"owner"* is set to the address of the user who created the contract. The *"lib"* variable should contain the address of the contract which will be invoked by delegatecall(). Now

, when any user invoke the function *delegatecalltest()* , the *callerContract* internally invokes the *"Lib"* contract function *pwn()* which is suppose to change variable **"owner"** of the *Lib* contract. But due to the *delegatCall()* the variable *owner* of the contract *Lib* is unchanged rather the variable **"owner"** of the *CallerContract* is changed to address of the callee. Advantage of using *delegatecall()* is that since target contract has the access of the caller contract, target contract can be used as a library where actual function logic can be written to update a variable in caller contract. However it can has some serious consequences if a code is not properly written.For example , the programmer may not want to change the variable *"owner"* but it can be changed by a delegate call without having a function extensively in the contract to change it.

Figure 2.4: Demonstration of working of Delegate Call

```
2   contract callercontract{
3       address public owner;
4       Lib public lib;
5       constructor(Lib _lib) public{
6           owner=msg.sender;
7
8       }
9       function delegatecalltest() public
10      {
11          address(lib).delegatecall(abi.encodeWithSignature("pwn()"));
12      }
13  }
14
15
16
17
18
19  //acts as library function
20  contract Lib{
21      address public owner;
22      function pwn() public{
23          owner=msg.sender;
24      }
25  }
```

Coming back to the parity wallets, the parity wallets was shivered by two attacks; the first attack cost around $30 million whereas second attack cost a huge $280 million.
Notice the use of *delegatecall()* throughout the *"Wallet"* contract to enable the use of shared libraries, saving precious storage space on the blockchain.

First parity attack:

There are few things things to observe carefully in the two contracts. Firstly The function *initWallet()* in the *"WalletLibrary"* is used to change the owner of the *"Wallet"*. Secondly the fallback function in line 15 of *wallet* contract is used deposit money in the wallet. In addition to depositing money, the fallback function is also used as some kind of error catching methodology

Figure 2.5: A snippet of the *"wallet"* contract

```
contract Wallet is WalletEvents {

  // WALLET CONSTRUCTOR
  //   calls the `initWallet` method of the Library in this context
  function Wallet(address[] _owners, uint _required, uint _daylimit) {
    // Signature of the Wallet Library's init function
    bytes4 sig = bytes4(sha3("initWallet(address[],uint256,uint256)"));
    address target = _walletLibrary;


  }

  // METHODS

  // gets called when no other function matches
  function() payable {
    // just being sent some cash?
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }

  // Gets an owner by 0-indexed position (using numOwners as the count)
  function getOwner(uint ownerIndex) constant returns (address) {
    return address(m_owners[ownerIndex + 1]);
  }

  // As return statement unavailable in fallback, explicit the method here

  function hasConfirmed(bytes32 _operation, address _owner) external constant returns (bool) {
    return _walletLibrary.delegatecall(msg.data);
  }

  function isOwner(address _addr) constant returns (bool) {
    return _walletLibrary.delegatecall(msg.data);
  }
```

Figure 2.6: A snippet of the *"WalletLibrary"* contract

```
    function initWallet(address[] _owners, uint _required, uint _daylimit) {
        initMultiowned(_owners, _required);
        initDaylimit(_daylimit) ;
    }

    // kills the contract sending everything to `_to`.
    function kill(address _to) onlymanyowners(sha3(msg.data)) {
        suicide(_to);
    }

    // the number of owners that must confirm the same operation before it is run.
    uint m_required;
    // pointer used to find a free slot in m_owners
    uint m_numOwners;

    uint public m_dailyLimit;
    uint public m_spentToday;
    uint public m_lastDay;

    // list of owners
    uint[256] m_owners;
    uint constant c_maxOwners = 250;

    // index on the list of owners to allow reverse lookup
    mapping(uint => uint) m_ownerIndex;
    // the ongoing operations.
    mapping(bytes32 => PendingState) m_pending;
```

in case the function invoked by the user does not matches any of the function syntax, it forwards the message to the *"WalletLibrary"* by making a delegatecall.

The *initwallet()* function is only called by the constructor during the contract creation where the creator provide the multiple addresses as arguments as in line 5. The interesting vulnerability in the code is that though the *initwallet()* function is supposed to be called only once during the contract creation, there was no safeguard to protect it from calling by a external contract.So, when an attacker calls Wallet.initWallet(*data**), Wallet's fallback function is triggered (Wallet does not implement an initWallet function), and the jump table lookup fails. Wallet's fallback function then delegatecalls *"WalletLibrary"*, forwarding all call data in the process. This call data consists of the method id for the *initWallet()* function and the attacker's address as argument. When WalletLibrary receives the call data, it finds that its initWallet function matches the function selector and runs initWallet(*data**) in the context of Wallet, setting Wallet's owner variable to attacker. BOOM! The attacker is now the wallet's owner and can withdraw any funds at his leisure.

Second parity attack

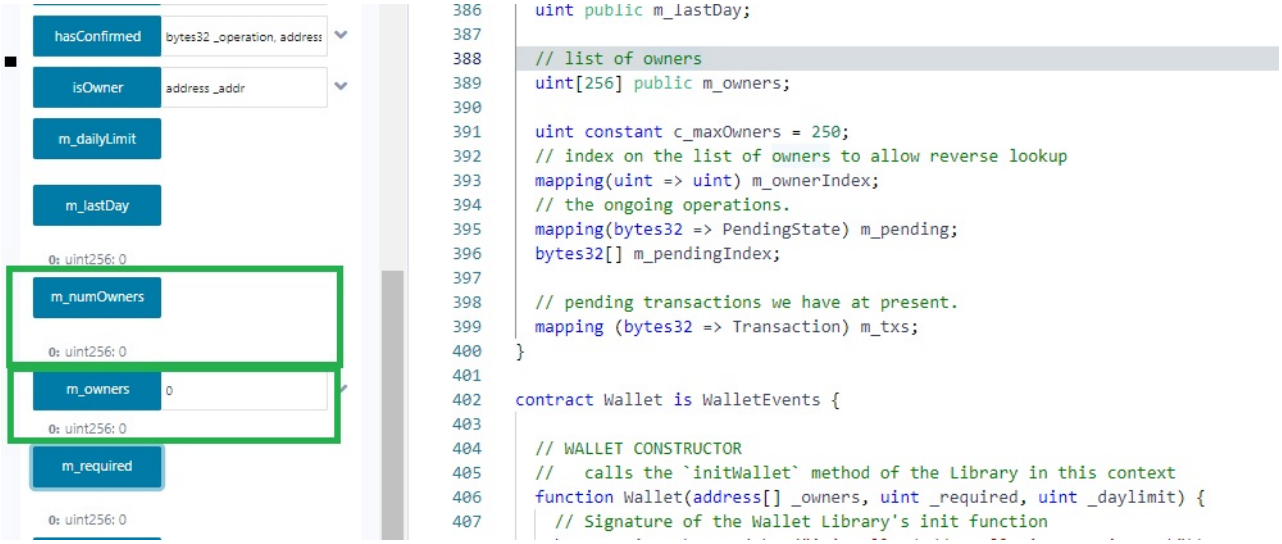
The first parity Lee et al. (2019) vulnerability was solved by introducing a **modifier** in the *initwallet()* function through which it was restricted to be invoked only at the time of creation and not by any external call. Now the second parity attackTechnologies (2017) was more of a lack

Figure 2.7: Introduction of modifier in the *initwaller()*

```
218  // throw unless the contract is not yet initialized.
219  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
220
221  // constructor - just pass on the owner array to the multiowned and
222  // the limit to daylimit
223  function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
224    initDaylimit(_daylimit);
225    initMultiowned(_owners, _required);
226  }
```

of developers security check of the code. The contract *"WalletLibrary"* was supposed to used only as Library Contract and not as contract of its own. This contract contains the same state variables as the *"Wallet"* contract and were left initialized. Thus when we invoke the variables of *"WalletLibrary"* contract, the values simply shows "0".This means there are no owner of the *"WalletLibrary"* contract This time the attack was more simple.The attacker simple had to initialize the variables and pass his own address.This is exactly what he did. He invoked

Figure 2.8: State variables of *"WalletLibrary* contract



the *initWallet()* function of the library contract by passing his own address in the ***"owners[]"*** array and **"0"** in the **_required** field which simply means that number of keys required for any function call is none. He could easily pass over the modifier because ***m_numOwners*** variable contains **"0"** initially. Now he invoked the ***kill()*** function. He could get pass over the **onlymanyOwners** modifier because he has set the number of signatures required to be **"0"**. Thus the *"LibraryWallet"* contract was killed and all the wallets that were using it as shared library for functionalities cannot further *delegateCall()* its functions. Because of it , all the Ethers in the wallets were frozen forever causing a huge damage of $250 millions

2.2.6 Unhandled Exception

In any smart contract in Ethereum, it often refers to another smart contract for various functionalities.In doing so it calls the other contract's address functions with reference to the contract's address.But there may be situations where called contracts function may fail and revert to its original state and return false to the caller contractPraitheeshan et al. (2019). There may be different reasons for the called function to fail namely not enough gas to execute the transaction, exceeding the caller stack limit and also occurring of some unexpected results and so on. When the called function is failed in the callee contract , exception is raised and this exception should be returned to the caller contract. The return value should be explicitly checked by the caller

contract to confirm whether the call has been successfully completed or not. However solidity is not uniform in dealing with this exceptions and there are seen inconsistencies in dealing with the returning the exception to the caller contract.

A attacker may cause the ***send()*** function to fail intentionally by invoking any functions in a contract.The limit of **Ethereum Virtual Machine** stack limit is **1024** frames. The call stack depth is increment by one when a function is called at once. At any situation , if the depth goes beyond 1024 , it raises a exception.An malicious user can intentionally arise this situation by simple calling its itself more than 1024 frames.

Figure 2.9: "Example of exception Handling errors

```
1 //Exception_exmaple.sol
2 contract Exception_exmaple {
3 address public currentInvestor;
4 uint public currentInvestment = 0;
5
6 function() payable public {
7 unit minimumInvestment =
currentInvestment * 11 / 10;
8 require(msg.value > minimumInvestment);
9
10 //document new investor
11 address previousInvestor =
currentInvestor;
12 currentInvestor = msg.sender;
13 currentInvestment = msg.value;
14
15 //payout previous investor
16 previousInvestor.send(msg.value);
 }
 }
```

A inconsistency is shown with help the contract given in Figure 2.7. The *Exception_example.sol* is a smart contract which pays the amount the interest to a user according to his amount of money he invested and the order the order in which investment is made. Here a malicious user can intentionally exploit the call-stack depth for personal gains. And he can purposefully increase the call-stack depth to 1023 frames which will lead to failure of payments by other investors. By doing so , he can make his interest payment earlier than other users.

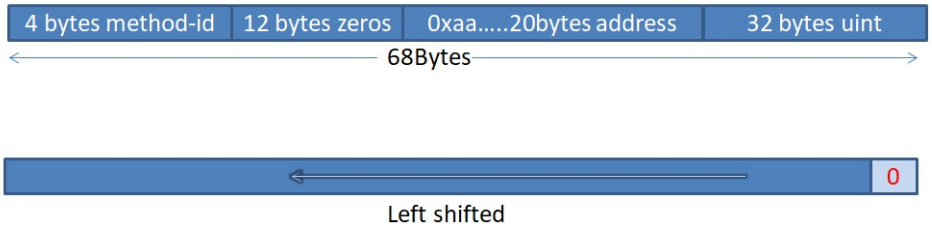
2.2.7 The short address attack

It is a bug discussed in the paper Torres et al. (2021) in the way methods of solidity contracts encode and decode argument.Though it was first observed in a **GOLEM** network, it can be applicable to every **ERC20** based contract. Raw Ethereum does not have methods or functions. They just have a bytecode of the contract which is written in a high-level language like **solidity** and they use the Ethereum contract ABI to specify how a contract bytecode divided into methods. ***Transfer()*** is a method of sending fund to some address from the caller account in the Golem network where **GNT** is the native currency. **"transfer(*address* a, *uint* v)"** method of sending ***"v"*** amount of GNT to address **0xa.....(20 bytes)** needs 3 pieces if information :

- 4 Bytes Method ID : 09059cbb
- 32 Bytes destination address where 20 bytes is the address with leading zeros the remaining
- 32 bytes being the fund to be transferred The combination of the above 3 information will be the full transaction in the bytecode.

Now if we leave anyone any bit from the address i.e, provide a short address , EVM will padd a**"0"** at the whole transation Bytecode rather than adding in the address portion of the bytecode. This can be dangerous because left shifting the whole bytecode may increase the transaction money by multiples of two

Figure 2.10: bytecode of the *transfer()* function



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.5;
contract vulnerable {
mapping(address => uint256) public balance;
function credit(address to) payable public {
    balance[msg.sender] += msg.value;
}
function check_balance () public
returns (uint) {
return balance[msg.sender];
}

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

(a)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.5;
import 'DAO.sol';
contract DAOAttacker {
        VulnerableContract public dao = DAO(0xDa32C9e....);
        address owner;
        constructor(DaoAttacker) public {
                owner = msg.sender;
        }
        function() attack public {
                dao.withdraw(dao.checkBalance(this));
                    }

        function() payable public{
                dao.withdraw(address(this).balance);
                }
}
```

(b)

Figure 2.11: (a) The Vulnerable DAO Contract (b) Attacker Contract

2.2.8 Re-entrancy vulnerability/attack

The DAO contract in 2016Praitheeshan et al. (2019) was the one where the reentrancy bug first came into existence. Here the withdraw() function of the vulnerable contract was recursively called again and again until all the gas was consumed. The withdraw() function of the vulnerable contract sent money to the attacker contract where its fallback function() is invoked. Now the attacker cleverly made the fallback function() in such a way that it again calls the withdraw() function of the vulnerable contract. The process repeats until the gas was consumed or all the vulnerable contract balance is withdrawn. A sample piece of code is described in the figure2.11

2.2.9 Unchecked Send()

The send() is an inbuilt function in **Solidity** used to send Ether from one address to another. Let us Consider a contract executing a send() operation to some address and due to some unavoidable reasons, the transaction fails. The execution does not stop and moves forward with

the next instruction without raising an error. Since the transaction is a successful one, the user may think that the money has been sent to the desired address.

Chapter 3

Xrpl Ledger And Hooks

3.1 Definition

The XRP Ledger (XRPL) Hooks are a powerful and innovative feature introduced to the XRP Ledger ecosystem, enabling smart contract-like functionality on the XRPL. Hooks are small, efficient WebAssembly (WASM) programs that can be attached to XRPL accounts, allowing users to implement custom logic and automate specific actions in response to incoming or outgoing transactions. One of the primary benefits of Hooks is that they provide a lightweight and efficient alternative to traditional intelligent contracts, which can be resource-intensive and slow. Hooks are designed to execute quickly and consume minimal resources, making them ideal for use on the XRPL, which prioritizes speed and scalability. Hooks can be used for a wide range of applications, including but not limited to multi-signature account management, automated escrow services, decentralized finance (DeFi) applications, and token issuance. By enabling users to create custom logic and automate actions, Hooks empower developers to build more complex and feature-rich applications on the XRPL. Security is a critical consideration in the design of Hooks. They are sandboxed, meaning they run in a secure and isolated environment,

preventing unauthorized access to sensitive account information or funds. Additionally, Hooks are deterministic, ensuring their execution produces consistent results across the network. The XRP Ledger (XRPL) Hooksxrp (????) introduce smart contract functionality to the XRPL, enabling layer one custom code to influence the behavior and flow of transactions. Hooks are small, efficient pieces of code attached to an XRPL account, allowing logic to be executed before and/or after transactions. This functionality can range from simple rules, such as rejecting payments below 10 XRP or automatically sending 10% of outgoing payments to a savings account, to more advanced operations. Hooks can also store small, simple data objects, enabling more complex logic. For example, a Hook could check if the sender of an incoming payment is on a list maintained by another Hook and reject the transaction if the sender is present. Notably, Hooks are not Turing-Complete, which, although often considered the gold standard for smart contracts, can be inappropriate due to potential resource consumption and security risks.

3.1.1 Hooks on outgoing txn

Hooks in the XRP Ledger are triggered by transactions, executing custom logic defined by the account owner. They provide users with greater control and flexibility in managing their XRPL accounts by automating actions, influencing the flow of transactions, and even revoking or modifying transactions based on specified conditions. When a transaction occurs, the Hook evaluates the conditions in its code and performs the corresponding actions if the conditions are met, which may include modifying or revoking the transaction. For example, Alice has a Hook on her XRPL account stating, "For all outgoing payments, checks if the same amount has been triggered twice. Additionally, if Alice had a Hook that checks for specific conditions, such as a maximum transaction amount, the Hook could modify or revoke the transaction if the conditions are not met. This demonstrates the power of Hooks in automating actions, customizing transaction behavior, and ensuring compliance with user-defined rules on the XRP Ledger.

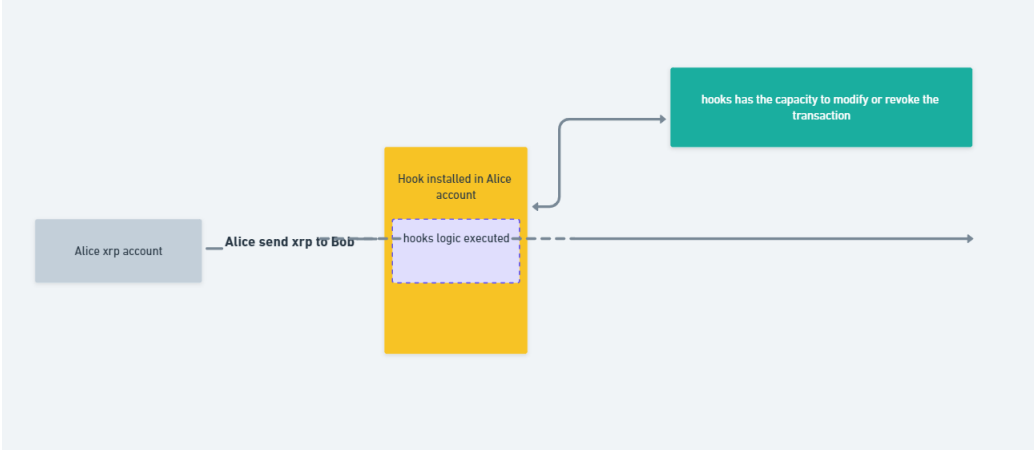


Figure 3.1: hooks triggered on outgoing transaction

3.1.2 Hooks triggered on incoming transaction

As in case of outgoing transactions, Hooks in the XRP Ledger can also be triggered by incoming transactions, allowing users to automate actions and customize transaction behavior based on specific conditions. For example, Bob has a Hook on his XRPL account that states, "For all incoming payments, send 10% to my savings account." When Alice sends 10 XRP to Bob, the Hook is triggered, and 1 XRP (10% of 10 XRP) is automatically sent to Bob's savings account, while the remaining 9 XRP is transferred to Bob's account. This demonstrates how Hooks can automate actions and influence the flow of transactions, providing users with greater control and flexibility in managing their XRPL accounts.

Figure 3.2: hooks triggered on incoming transaction

3.1.3 How to set up hooks

Hook web assembly bytecode is installed onto an XRPL account using the ***SetHook*** transaction. This sethook transaction is attaching a Hook to the XRPL account with the address

```
{
    Account: "r4GDFMLGJUKMjNhhycgt2d5LXCdXzCYPoc",
    TransactionType: "SetHook",
    Fee: "2000000",
    Hooks:
    [
        {
            Hook: {
                CreateCode: fs.readFileSync('accept.wasm').toString('hex').toUpperCase(),
                HookOn: '0000000000000000',
                HookNamespace: addr.codec.sha256('accept').toString('hex').toUpperCase(),
                HookApiVersion: 0
            }
        }
    ]
}
```

Figure 3.3: ***SetHook*** transaction

"r4GDFMLGJUKMjNhhycgt2d5LXCdXzCYPoc". The Hook is defined by the WebAssembly (WASM) code in the file "accept.wasm", which is read and converted to hexadecimal format using the fs.readFileSync() function. The Hook is set to trigger on all transactions, as indicated by the "HookOn" field being set to "0000000000000000". The Hook's namespace is defined by the SHA-256 hash of the string "accept", which is converted to hexadecimal format using the addr.codec.sha256() function. The Hook's API version is set to 0. The transaction fee is set to 2,000,000 drops, and the Hooks field is an array containing a single object that defines the Hook's details. Once this sethook transaction is signed and submitted to the XRPL network, the Hook will be attached to the specified account and can be triggered by incoming and outgoing transactions.

3.1.4 Security Model of XRPL Hooks

XRPL Hooks are executed in a fast, light, and secure virtual machine based on Web Assembly, a concise and portable binary executable format. This format is compatible with any W3C standards-compliant web assembly runtime, ensuring broad compatibility and adherence to web standards. The core security model of XRPL Hooks is similar to an operating system, with the

web assembly runtime environment (xrpld) acting as the OS and the Web Assembly binary (hook) as a user-space application. Hooks are sandboxed by xrpld, allowing them to modify their own memory and call functions explicitly exposed by xrpld. Hooks cannot call external or internal processes, services, or endpoints, and can only access the Hook API. They cannot modify or read any part of the system outside their well-defined and pre-allocated memory. The hook's memory is limited by a validator-votable figure, nominally 64kib. When Hooks request actions, such as blocking a transaction, they provide pointers within their memory-space for xrpld to read and write. If incorrect pointers are provided, Hooks can only damage their own memory, ensuring the surrounding system remains unaffected. This design ensures a secure and efficient environment for XRPL Hooks to operate within.

3.1.5 Limitions of Xrpl Hooks

XRPL Hooks are a scripting system that enables developers to create decentralized applications on the XRP Ledger. However, they have some limitations compared to other scripting systems like Ethereum Smart Contracts. For example, XRPL Hooks can only be triggered on sending and receiving transactions, and they cannot directly interact with other Hooks on the XRP Ledger. Additionally, there is a limit of 10 Hooks per account, which may limit the complexity of applications that can be built. These limitations could potentially make it more challenging to build complex, interconnected applications on the XRP Ledger compared to other blockchain platforms.

Limited Interaction: The inability of hooks to directly interact with other hooks on the XRP Ledger restricts the complexity and sophistication of decentralized applications (dApps) that can be built. It may limit the development of complex smart contract-like functionalities that require direct interaction between multiple hooks.

Reduced Flexibility: The limited trigger points for hooks (sending and receiving transactions) may restrict the range of use cases that can be implemented. Certain types of applications, such as those requiring periodic or time-based triggers, may not be feasible or may require workarounds.

Scalability Concerns: If a single account is limited to only 10 hooks, it can potentially

limit the scalability of applications that require multiple hooks to be deployed. Developers may need to carefully manage and optimize their hook usage, which could impact the design and functionality of their applications.

Dependency on External Events: Since hooks can only interact indirectly when transactions are sent through hooks and the receiver's hook is triggered, the execution of logic within hooks is dependent on external events and the actions of other participants. This may introduce complexities and potential vulnerabilities, as the expected behavior of other participants cannot be guaranteed.

Development and Debugging Challenges: The limitations of XRPL Hooks may introduce additional challenges for developers. The reduced flexibility and limited interaction may require more complex coding techniques or workarounds to achieve desired functionalities. Debugging and testing hooks in such scenarios may also be more complex.

Despite these disadvantages, the XRP Ledger has a number of potential advantages. The XRP Ledger is one of the fastest blockchain platforms available, with transactions being processed in seconds. Transactions on the XRP Ledger are very low cost, making it a good option for payments and other financial applications. The XRP Ledger is a very energy-efficient blockchain platform, making it a good choice for environmentally conscious businesses.

3.1.5.1 Comparison with Ethereum Smart Contracts

	XRPL Hooks	Ethereum Smart Contracts
Trigger points	Primarily triggered by sending and receiving transactions on the XRP Ledger	Can be triggered by various events, such as external transactions, time-based triggers, or interactions with other smart contracts
Direct interaction	Cannot directly interact with each other on the XRP Ledger	Can directly interact with each other on the Ethereum blockchain
Programming language and flexibility	Language independent but intentionally kept not turing-complete due to security reasons	Written in Solidity, a Turing-complete programming language
Contract deployment and gas fees	Associated with individual accounts on the XRP Ledger, limited to 10 hooks per account	Need to be deployed on the blockchain, gas fees for every computational step
Ecosystem and tooling	Smaller ecosystem and limited tooling	Mature ecosystem, wide range of tools, libraries, and frameworks

Chapter 4

Proof Carrying Hooks in XRPL

4.1 What is PCC?

Smart Contracts commonly experience bugs and errors when the code deviates from its intended behavior. One of the major challenges with Blockchain is the arduous task of updating the code, even when a bug is detected. However, there is a solution to this predicament called **Proof Carrying Code (PCC)**Necula (1997). The concept of **PCC** involves keeping a separate code alongside the original smart contract, which serves as a proof and can be employed to verify the correctness and accuracy of the smart contract. By utilizing PCC, developers can ensure a higher level of confidence in the functioning of their smart contracts. Smart contracts operate within a defined set of states for each functionality, enabling the tracking of state changes to ensure correctness. As external entities interact with the smart contract exclusively through its methods, it becomes crucial to monitor and verify the state changes following each function call. To achieve this verification, the producer is responsible for specifying the expected end state resulting from each function call, which will be formally defined. This approach enhances the transparency and accountability of smart contract operations.

The Proof Carrying Code (PCC) encompasses several key elements, including:

Method Specification: This involves providing a detailed specification for each method in terms of the states it operates on and the variables it utilizes. By clearly defining these aspects, the PCC ensures a thorough understanding of the functionality.

Formal Definition of State Changes: For every function within the smart contract, the PCC establishes a formal definition of the expected state changes. This enables rigorous verification of the correctness of state transitions.

Controlled External Interactions: In cases where the smart contract interacts with external contracts, the PCC treats such interactions as ***coroutines***. It imposes specific guard conditions that govern when and how the contract can interact with external entities, ensuring secure and controlled integration.

4.2 PCC using the invariant of a hook

An example of **XRPL** hooks which contains the transaction details in JSON format and the hook logic :

Example1

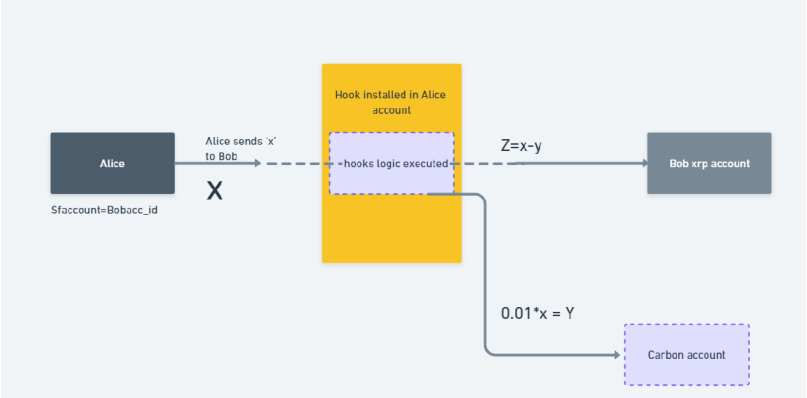


Figure 4.1: A hook transaction

```c
#include <stdint.h>
#include "../hookapi.h"
int64_t hook(int64_t reserved ) {

    TRACESTR("Carbon: started");
    // this api fetches the AccountID of the account the hook currently executing is installed on
    // since hooks can be triggered by both incoming and ougoing transactions this is important to know
    unsigned char hook_accid[20];
    hook_account((uint32_t)hook_accid, 20);

    // next fetch the sfAccount field from the originating transaction
    uint8_t account_field[20];
    otxn_field(SBUF(account_field), sfAccount);
    // compare the "From Account" (sfAccount) on the transaction with the account the hook is running on
    int equal = 0;
    BUFFER_EQUAL(equal, hook_accid, account_field, 20);
    if (!equal)
    {
        // if the accounts are not equal (memcmp != 0) the otxn was sent to the hook account by someone else
        // accept() it and end the hook execution here
        accept(SBUF("Carbon: Incoming transaction"), 2);
    }

    // execution to here means the user has sent a valid transaction FROM the account the hook is installed on

    // fetch the sent Amount
    // Amounts can be 384 bits or 64 bits. If the Amount is an XRP value it will be 64 bits.
    unsigned char amount_buffer[48];
    otxn_field(SBUF(amount_buffer), sfAmount);
    int64_t drops_to_send; // this will be the default
    int64_t otxn_drops = AMOUNT_TO_DROPS(amount_buffer);


    drops_to_send = (int64_t)((double)otxn_drops * 0.01f); //


    // hooks communicate accounts via the 20 byte account ID, this can be generated from an raddr like so
    // a more efficient way to do this is precompute the account-id from the raddr (if the raddr never changes)
    uint8_t carbon_accid[20];
    int64_t ret = util_accid(
            SBUF(carbon_accid),                                 /* <-- generate into this buffer  */
            SBUF("rfCarbonVNTuXckX6x2qTMFmFSnm6dEWGX") );       /* <-- from this r-addr           */
    TRACEVAR(ret);
// fees for emitted transactions are based on how many txn your hook is emitted, whether or not this triggering
int64_t fee_base = etxn_fee_base(PREPARE_PAYMENT_SIMPLE_SIZE);

// create a buffer to write the emitted transaction into
unsigned char tx[PREPARE_PAYMENT_SIMPLE_SIZE];

// we will use an XRP payment macro, this will populate the buffer with a serialized binary transaction
PREPARE_PAYMENT_SIMPLE(tx, drops_to_send++, fee_base, carbon_accid, 0, 0);

// emit the transaction
emit(SBUF(tx));

// accept and allow the original transaction through
accept(SBUF("Carbon: Emitted transaction"), 0);
return 0;
```

Figure 4.2: Example_1

The hook function does the following:

1. Fetch the account_id of the xrp transaction using the `hook_account` function and store it in the `hook_accid` array.

2. Retrieve the value of the "sfAccount" field from the xrp transaction using the `otxn_field` function and store it in the `account_field` array.

3. Compare the `hook_accid` and `account_field` arrays to check if the transaction was sent to the hook account by the same account(to itself)

4. If the accounts are not equal, accept the transaction with a status code of 2 using the `accept` function and end the hook execution.

5. If the accounts are equal, proceed with the execution, indicating that the transaction is sent from the hook account.

6. Fetch the sent amount from the transaction using the `otxn_field` function and store it in the `amount_buffer` array.

7. Convert the amount in `amount_buffer` to drops using the `AMOUNT_TO_DROPS` macro and assign it to the `otxn_drops` variable.

8. Calculate the `drops_to_send` value as 1% of the `otxn_drops` value.

9. Generate the AccountID for the "rfCarbonVNTuXckX6x2qTMFmFSnm6dEWGX" raddr using the `util_accid` function and store it in the `carbon_accid` array.

10. Create a buffer `tx` to write the emitted transaction into, populate it with a serialized binary transaction using the `PREPARE_PAYMENT_SIMPLE` macro and the relevant parameters, emit the transaction using the `emit` function, and accept the original transaction to allow it to proceed with a status code of 0 using the `accept` function.

Invariant for the above hook

```
X= amount sent by alice
z= amount send to Bob
y= amount send to carbon account
{y=0.01*x
z=x-y}
Invar: x=z+y
```

Figure 4.3

```c
#include <stdint.h>
#include "../hookapi.h"
int64_t hook(int64_t reserved ) {
    // fetch hook account id
    uint8_t hook_accid[20];
    hook_account(SBUF(hook_accid)
    // next fetch the sfAccount field from the originating transaction
    uint8_t account_field[20];
    otxn_field(SBUF(account_field), sfAccount);

    // compare the "From Account" (sfAccount) on the transaction with the account the hook is running on
    int equal = 0; BUFFER_EQUAL(equal, hook_accid, account_field, 20);
    if (equal)
    {
        accept(SBUF("Doubler: Outgoing transaction. Passing."), 2);
        return 0;
    }

    uint8_t digest[96];  //fetch last closed ledger hash
    ledger_last_hash(digest, 32);

    uint8_t hash[32];
    util_sha512h(SBUF(hash), SBUF(digest)  //take the hash of the digest

    // first digit of lcl hash is our biased coin flip, you lose 60% of the time :P
    if (hash[0] % 10 < 6)
        accept(SBUF("Doubler: Tails, you lose"), 4);


    unsigned char amount_buffer[48];
    otxn_field(SBUF(amount_buffer), sfAmount);   // fetch the sent Amount in amount_buffer
    int64_t drops_to_send = AMOUNT_TO_DROPS(amount_buffer) * 2; // doubler pays back 2x received

    int64_t fee_base = etxn_fee_base(PREPARE_PAYMENT_SIMPLE_SIZE);
    uint8_t tx[PREPARE_PAYMENT_SIMPLE_SIZE];

    // we will use an XRP payment macro, this will populate the buffer with a serialized binary transaction
    // Parameter list: ( buf_out, drops_amount, drops_fee, to_address, dest_tag, src_tag )
    PREPARE_PAYMENT_SIMPLE(tx, drops_to_send, fee_base, account_field, 0, 0);


    // emit the transaction
    emit(SBUF(tx));

    // accept and allow the original transaction through
    accept(SBUF("Doubler: Heads, you won! Funds emitted!"), 0);
    return 0;

}
```

Figure 4.4: Doubler hook

Example_2: Doubler hook

The hook function does the following:

1. Fetch the account_id and "sfaccount" id and compare the both.
2. If the accounts are equal, proceed with the execution, indicating that the transaction is sent from the hook account.
3. The hash of the last closed ledger is fetched and stored in the `digest` array.The SHA-512 hash of the digest array is computed and stored in the `hash` array
4. The first digit of the computed hash is used as a biased coin flip.The coin flip indicates tails if the digit is less than 6, the hook accepts the transaction and ends.
5. The amount field of the originating transaction is fetched and stored in the `amount_buffer` array. The amount is converted to drops and Doubled to determine the amount to be sent
6. The base fee for the emitted transaction is determined. A transaction buffer tx is created and populated with a serialized binary transaction using the provided macros. The transaction is emitted using the emit function.

```
X= original amount
Y= amount to be send to Y
Invar: x= 2*y
```

Figure 4.5: Invariant for the above

Chapter 5

Co-routine structure for smart contracts

5.1 What are Co-routines?

A coroutine, is a programming construct that allows a function to be paused and resumed later, without losing its state. It enables a program to have multiple entry points or entry points that can be interrupted and resumed at specific locations. Coroutines provide a way to write more efficient and readable code for certain types of tasks that involve managing concurrent or asynchronous operations. Unlike regular functions, which start from the beginning and execute until they reach a return statement, coroutines can be suspended at specific points and resumed later, picking up from where they left off. While you can only call a function and return from it, you can call a coroutine, suspend and resume it, and destroy a suspended coroutine.

By incorporating coroutines into the contract's design, various patterns like cause-effect and emergency stop can be applied, offering greater control over the contract's behavior. This enables the implementation of conditions before control is shifted and after it returns, ensuring proper execution of the smart contract. For instance, consider the infamous DAO attack where the called contract exhibited malicious behavior. In this scenario, the calling contract

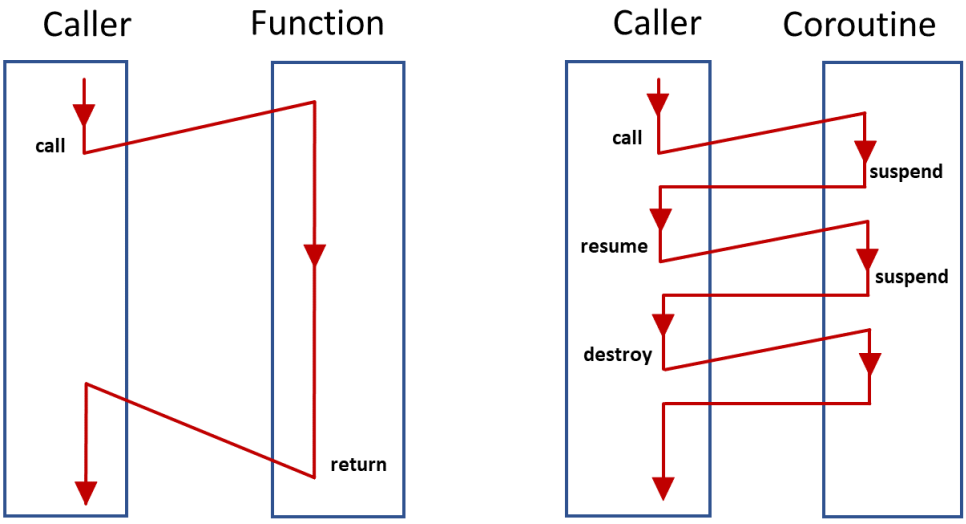


Figure 5.1: co-routine vs function call

had no control over the fallback function() of the called contract, allowing reentry into the calling contract. However, by leveraging coroutines, such unwanted behavior can be prevented by enforcing necessary conditions and checks. By abstracting coroutine , a smart contract can effectively manage control flow and prevent potential vulnerabilities or undesired interactions between contracts. The coroutine construct enables the implementation of structured and controlled execution paths, enhancing the security and reliability of smart contracts.

5.1.1 Formalizing co-routines correctness

The main difference between a function call and a coroutine call is that coroutines have multiple entry and exit points. In the context of smart contracts, a function call is typically under the control of the caller contract. However, problems can arise when two contracts interact, as the caller contract may not always have complete control over the called contract. To address this, we can leverage the concept of coroutines. In the case of smart contracts, we can consider coroutines when calling or interacting with another contract. However, before using coroutines, there are several points that need to be addressed:

COMMUNICATION LIST: This refers to the shared variables through which interaction is made. Information can only pass through these variables, and they serve as a means of communication between the contracts.

CONTRACT ADDRESS: It is important to have the address of the contract with which interaction is taking place. This allows for proper identification and targeting of the desired contract.

GUARD CONDITIONS: These are conditions that need to be evaluated before and after each control shift in the coroutine. Guard conditions can take various patterns, such as the cause-interaction-effect pattern, emergency stop pattern, or speed bump pattern. These conditions help ensure proper control and security during the interaction.

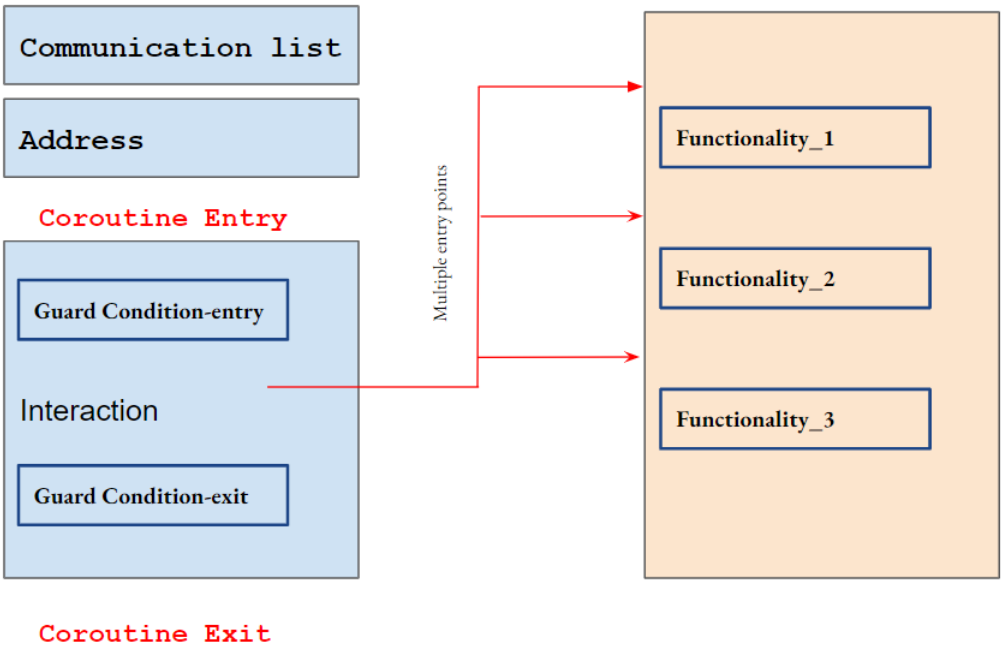


Figure 5.2: co-routine steps

By considering these points and implementing coroutines in smart contracts, we can enhance control and coordination between interacting contracts, leading to more robust and secure smart contract systems.

The functionality of the coroutine follows a series of steps:

Step 1: **Initialization of Communication List**: The communication list, which serves as the shared variable for interaction, is initialized. This step sets up the necessary data structure for communication between coroutines.

Step 2: Resolving Addresses: The addresses of the relevant contracts or entities involved

in the coroutine interaction are resolved. This ensures that the communication is directed to the intended recipient.

Step 3: Entry: The coroutine enters its execution flow, starting from a designated entry point. This is the initial step in the coroutine's execution path.

Step 4: Checking Pre-Condition: Before any interaction takes place, the coroutine checks the pre-condition to ensure that the required conditions for the interaction are met. This step ensures that the interaction is performed under the desired circumstances.

Step 5: Interaction: The actual interaction between the coroutine and the target contract or entity occurs. This can involve data exchange, function calls, or any other form of communication specified by the coroutine.

Step 6: Checking Post-Condition: After the interaction, the coroutine verifies the post-condition to validate the outcome of the interaction. This step ensures that the desired effects or changes resulting from the interaction have been achieved.

Step 7: Exit: The coroutine exits its execution flow, reaching an appropriate exit point. This marks the end of the coroutine's execution and its contribution to the overall system behavior.

By following these steps, the coroutine can effectively manage communication and interaction within a system, providing structured control and facilitating secure and coordinated execution paths.

5.1.2 Illustrating capturing of "Patterns" in Smart Contracts via Coroutines

5.1.3 Checks-Effects-Interaction pattern

The Checks-Effects-Interactions patternWohrer and Zdun (2018) is an important coding principle for functions. It helps us organize the code to avoid side effects and unwanted behavior. The pattern consists of three steps: checking preconditions, making changes to the contract's state,

and interacting with other contracts. That's why it's called the "Checks-Effects-Interactions Pattern." Following this principle, it is recommended to perform interactions with other contracts as the final step in any function, if possible.The reason for placing interactions with other contracts as the last step in any function is to prevent potential security vulnerabilities. When a contract interacts with another contract, such as transferring Ether, it effectively hands over control to that contract. This allows the called contract to execute potentially harmful actions. One example is the re-entrancy attack, where the called contract calls back the current contract before the initial invocation of the function containing the call is finished. This can lead to unwanted execution behavior, such as modifying state variables to unexpected values or causing operations like sending funds to be performed multiple times. By following the Checks-Effects-Interactions pattern and placing interactions as the final step, we minimize the risk of these security issues. However inorder to use this pattern in coroutines we need to guard it with pre-conditions and post-conditons

5.1.4 Emergency stop Pattern

In order to respond quickly to potential bugs or attacks that may arise in a contract, it is advisable to include emergency stops or circuit breakersWohrer and Zdun (2018). These mechanisms allow for the suspension of contract execution or specific parts of it when certain conditions are met. This helps to mitigate any potential risks and ensure the safety of the contract and its users. A recommended scenario would involve detecting a bug and promptly halting all critical functions, while still allowing users to withdraw their funds. The ability to trigger an emergency stop in a contract can be implemented in one of two ways. Firstly, it can be granted to a specific party, such as the contract owner or an authorized administrator. This designated party would have the authority to initiate the emergency stop when necessary. Alternatively, an emergency stop mechanism can be implemented through a predefined set of rules. In this approach, specific conditions or criteria would be established, and if those conditions are met, the contract would automatically initiate the emergency stop. Both methods provide a means to swiftly respond to potential issues and safeguard the contract's functionality and user funds.

5.2 Advantages of using co-routine

Here are the advantages of using coroutines in smart contract interaction:

1. Provides control over behavior during contract transitions: Coroutines offer a structured and explicit way to control the behavior when transitioning from one contract to another. Developers can define specific actions, conditions, or checks to be performed before, during, or after the transition. This level of control enables the implementation of custom logic and ensures desired behaviors are enforced during contract interactions.

2. Transactions can be rolled back if guard conditions are not satisfied: Coroutines allow for the implementation of guard conditions that need to be met before proceeding with a contract interaction. If these guard conditions are not satisfied, the transaction can be rolled back, preventing any undesired state changes. This rollback capability provides an additional layer of safety and allows for the enforcement of specific preconditions before executing the interaction.

3. Mitigates vulnerabilities like reentrancy by avoiding recursive calls: One of the vulnerabilities in smart contracts is the risk of reentrancy attacks. Coroutines can help mitigate this vulnerability by avoiding recursive function calls within a contract interaction. By controlling the execution flow using coroutines, developers can ensure that the contract completes its intended actions before allowing external calls to reenter the contract. This prevents unauthorized or unintended access to contract state during critical operations.

5.3 Smart Contracts abstracted as coroutines

A coroutine programming paradigm paves a way to capture different vulnerabilities in smart contract. Its provides a way to gain control over the undeterministic interaction between contracts. A illustrustation of few examples is shown below

5.3.1 Illustrating reentrancy vulnerability as co-routine structure

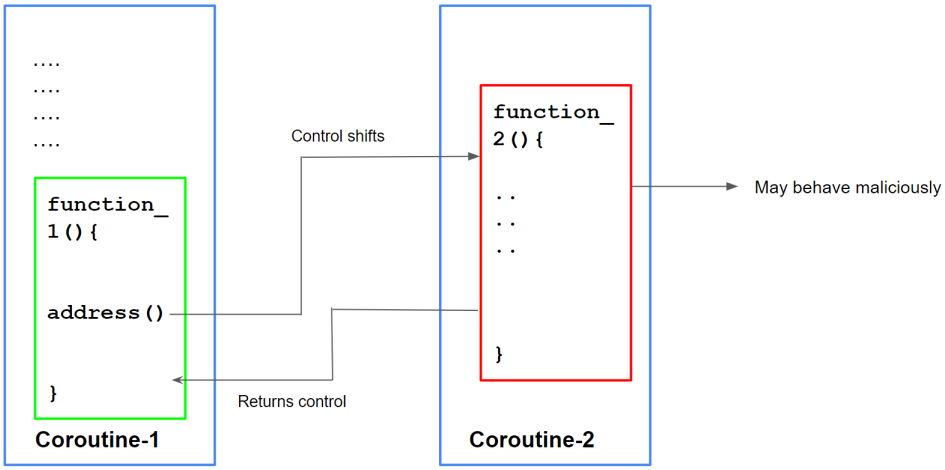


Figure 5.3: Possible reentrancy

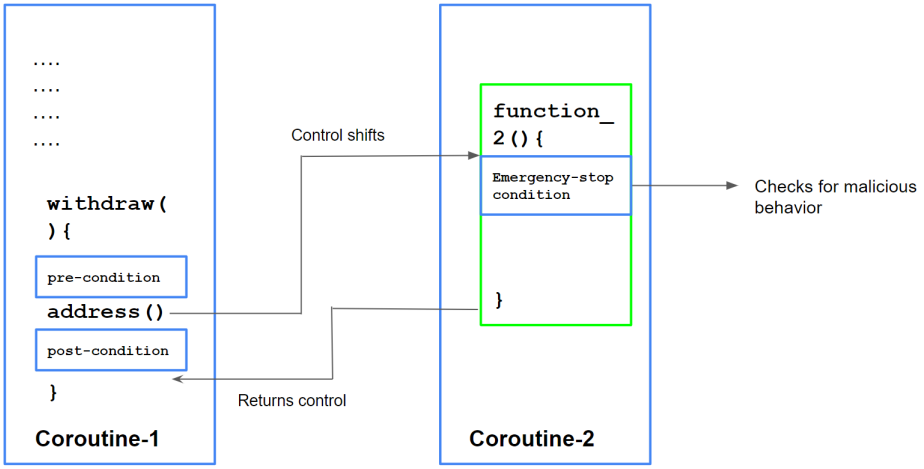


Figure 5.4: coroutine

In the first scenario depicted in the figure 5.3, a standard interaction is shown where the *function_2* of the called contract could potentially exhibit malicious behavior, which the caller contract has no control over. This lack of control over the behavior of the called contract can lead to vulnerabilities.

In contrast, in the second scenario illustrated in the figure 5.4, each program is treated as a coroutine, resulting in improved control over the shifting of control between contracts.

By adopting this approach, we enhance control flow management and security. To achieve this, the caller contract is designed using the cause-effect pattern, ensuring that the called contract behaves in the intended manner. Additionally, an emergency stop() guard is implemented within the fallback function(). This guard imposes certain conditions that must be satisfied for the transaction to proceed. If these conditions are not met, the transaction is immediately revoked, and control is returned to the caller contract.

Moreover by definition coroutines are not recursive. Thus there arises no question of reentrancy attack or any kind of attack involving recursive interaction between contracts.

5.3.2 Illustrating suicidal vulnerability as co-routine structure

When sending money to the SelfDestructContract from another contract, it's important to consider that the sending contract might not have information about the current status of the SelfDestructContract. If the SelfDestructContract has already been destroyed or self-destructed, any interaction with it, including sending funds, will result in the loss of those funds. Therefore, it's crucial to ensure that the SelfDestructContract is active and functioning as expected before initiating any transactions with it, he might loose it forever. Using coroutine structure we can have pre-condition to check the liveliness of the contract as shown in fig 5.3.2.

```
pragma solidity ^0.8.0;

contract SenderContract {
    address public receiverContractAddress;
    SelfDestructContract public receiverContract;

    constructor(address _receiverContractAddress) {
        receiverContractAddress = _receiverContractAddress;
        receiverContract = SelfDestructContract(receiverContractAddress);
    }

    function sendFundsToReceiverContract() external payable {
        // Forward the received funds to the receiver contract
        receiverContract.receiveFunds{value: msg.value}();
    }
}
```

```
pragma solidity ^0.8.0;

contract SelfDestructContract {
    address payable public beneficiary;
    uint public balance;

    constructor(address payable _beneficiary) {
        beneficiary = _beneficiary;
    }

    // Function to destroy the contract and transfer remaining funds to
    function destroy() public {
        require(msg.sender == beneficiary, "Only the beneficiary can ca
        selfdestruct(beneficiary);
    }

    // Function to receive funds
    function receiveFunds() external payable {
        // Increment the balance by the received amount
        balance += msg.value;
        // Optional: You can add custom logic here when receiving funds
    }

    // Function to withdraw funds by the beneficiary
    function withdrawFunds(uint amount) external {
        require(msg.sender == beneficiary, "Only the beneficiary can wi
        require(amount <= balance, "Insufficient balance");

        // Deduct the requested amount from the balance
        balance -= amount;
        // Transfer the requested amount to the beneficiary
        beneficiary.transfer(amount);
    }
}
```

Figure 5.5: Possible suicidal

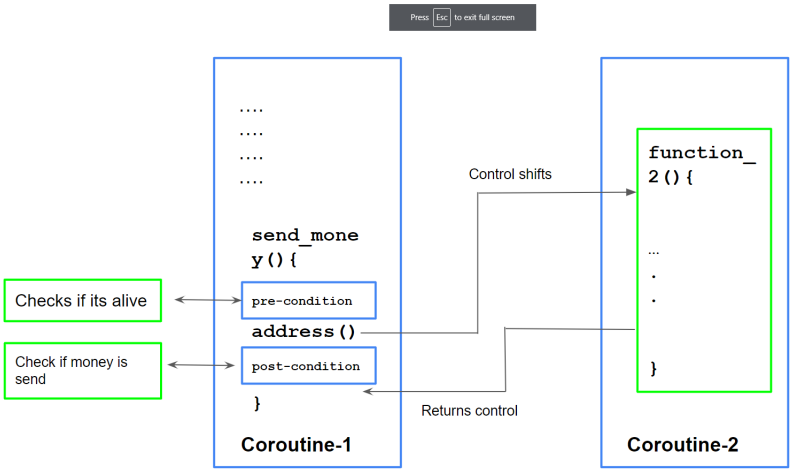


Figure 5.6: Co-routine capturing suicidal

5.4 Coroutine syntax

The co-routine syntax must include a communication list(list of shared variables), a pre-condition(p), interaction details , post-post condition(q)

$$< communication\_list >< p >< interaction(address) >< q > \tag{5.1}$$

The syntax of rentrancy vulnerability 5.3.1 would be:

```
<COMMUNICATION-LIST>
        List of global variables used in the contract
    {msg.sender,msg.value}
<p>
        Precondition
        Specify the condition that must be satisfied before the coroutine body executes
    amount>0

<interaction, (address)>

        // Specify the address of the other co-routines

    address(coroutine B)

<q>
        Postcondition
        Specify the condition that must be satisfied affter the coroutine body executes

  balances[msg.sender]-=amount
```

Figure 5.7: co-routine syntax to capture reentrancys

Chapter 6

Conclusion and Future Work

The findings and observations presented in this thesis lay the foundation for enhancing the application of smart contracts within the blockchain environment. This dissertation has delved into several significant vulnerabilities present in Solidity smart contracts, such as reentrancy and parity issues, and has provided effective methods to mitigate these risks. By introducing the concept of Proof-Carrying Code (PCC) for XRPL ledger hooks, we have demonstrated how the enforcement of PCC can enhance the integrity of smart contracts utilizing hooks, instilling confidence in their usage. Furthermore, we have explored the viability of utilizing co-routine structures for specifying smart contract integrity. The analysis has revealed that co-routines offer a promising approach to addressing vulnerabilities and ensuring the robustness of smart contracts. Through the illustration of various vulnerabilities, it has become evident that co-routines present a natural solution for mitigating risks and improving the overall security of smart contract implementations. However, it is important to note that the current architecture of platforms such as Ethereum or XRPL Ledger does not directly support coroutines. To incorporate coroutine structures, a dedicated compiler or a third-party conversion tool is required to convert coroutine code into Ethereum binaries or compatible formats for other platforms.

References

, ???? Xrpl hooks documentation. https://xrpl-hooks.readme.io/docs, accessed: June 24, 2023.

Lee, J. H., et al., 2019. Systematic approach to analyzing security and vulnerabilities of blockchain systems. Ph.D. thesis, Massachusetts Institute of Technology.

Nakamoto, S., 2008. Bitcoin whitepaper. URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019).

Necula, G. C., 1997. Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106–119.

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference. pp. 653–663.

Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R., 2019. Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605.

Technologies, P., November 2017. A postmortem on the parity multi-sig library self-destruct. URL https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-

Torres, C. F., Iannillo, A. K., Gervais, A., State, R., 2021. The eye of horus: Spotting and analyzing attacks on ethereum smart contracts. arXiv preprint arXiv:2101.06204.

Wohrer, M., Zdun, U., 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, pp. 2–8.

Acknowledgments

I wish to record a deep sense of gratitude to **Prof. Virendra Singh** & **Prof. R.K Shyamasundar**, my supervisor for his valuable guidance and constant support at all stages of my M.Tech study and related research.