DISTRIBUTED COMPUTING

Research Paper

Smart Security in Smart Contract

Supervisor : Dr. Kashif Mehboob

Batch: 2019

Year: Final Year

Department: Software Engineering

Project Group Members:

Name: Fizza Mariyam
Roll No: BESE-19002
Email: zahid4201745@cloud.neduet.edu.pk

Name: Abdul Moiz
Roll No: BESE-19022
Email: moiz4208342@cloud.neduet.edu.pk

Name: Kabeer Ahmed
Roll No: BESE-19028
Email: ahmed4201750@cloud.neduet.edu.pk

Smart Security in Smart Contract

Fizza Mariyam [1], Abdul Moiz [2] ,Kabeer Ahmed [3]
[1,2,3] Department of Software Engineering, NED University of Engineering & Technology, Karachi, Pakistan

***Abstract*-Smart contracts are contracts for any kind of agreement, but what makes them smart is that they run as code automatically on a blockchain. They implement the blockchain's decentralization idea by automating transactions in the absence of any centralized control. Although smart contracts have advanced quickly, there are still a number of security issues that need to be explored. Loss in money is frequently caused by such weaknesses. In the present world, a fully automated solution to vulnerability detection is desired. We summarize the security challenges of smart contracts. It is merely a human-made, error-prone computer application. The majority of smart contract flaws are caused by security issues in lines of computer code. Future scholars will receive full analyses and recommendations from our prestigious work. As we proposed methods of securing smart-contract through different approaches by taking in place of the secure coding practices, we analyze the success rate is much higher than relying on the tool. Our aggressive analysis of vulnerability in a smart contract depicted that complex logic sometimes provides a foothold to the attacker, the main point is to remediate it before anyone can exploit it or increase the severity. We examine the methods and tools that developers can use to safeguard data against hacking.**

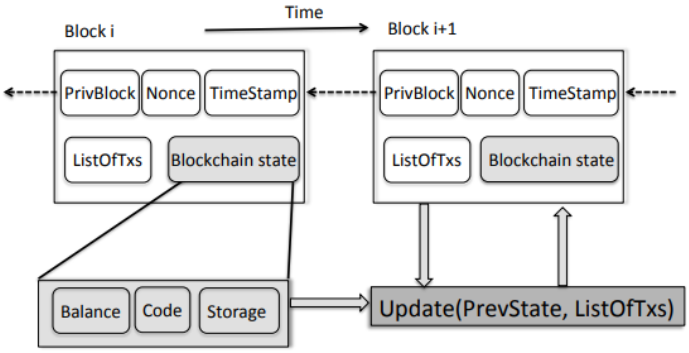
Keywords— smart contract; Ethereum; SmartCheck, security concerns; vulnerabilities; Xpath

I. Introduction

Blockchain was originally described in 1991 as was used for securing original digital documents. This technology remained unused for many years until in 2009 Satoshi Nakamoto and the first ever digital cryptocurrency was developed called Bitcoin. The blockchain technology has a property that once some data is recorded in them, it is difficult to alter them. The unique hash of every block is stored in the next block. The hash is created at the time of storing of data and if the data is changed its hash is also changed. All blocks contain the hash of previous blocks. So, if any block is tempered by all other blocks (nodes) in the network. Instead of a central server, Block chain uses peer-to-peer connection.

This technology is constantly evolving. One of its developments is the smart contracts. It means the code that runs on blockchain and it ensures an agreement between two parties. They are like normal contracts except they are digital and are programmed in a special programming language called solidity. We don’t need to rely on third parties. It's just you and the person who is receiving your transactions within the decentralized networks of blockchain. Smart contracts can hold all the received funds until a

certain goal is achieved. If the goal is achieved the money is transferred to the creator, if it fails then it is sent back to senders. But why should we trust smart contracts? Because it is immutable and distributed, means that it is not possible to alter the smart contract. Secondly, it is validated by everyone in the network. A part of crowdfunding, the smart contracts are used in insurance claims, postal to deliver payments, electronic voting systems etc. Nowadays there are a number of block chains that support smart contracts. The most popular among them is the Ethereum block chain. It is developed for creating smart contracts.



The design in famous cryptocurrencies like Ethereum. Every block is made up of several transactions.

But in spite of a number of security measures there is also a risk of detrimental attacks. A wide range of attacks are made on blockchain technologies related to cryptocurrency, E-wallets, transactions, smart contracts etc. Some famous attacks on smart contracts are DAO attack and King of Ether Throne. So, the security factor should be properly focused to avoid any kind of vulnerabilities.

There are some tools available such as Oyente, Osiris, Remix. In this paper we discuss a security tool called SmartCheck. Its performance is more accurate as compared to the others. It is a tool for checking the security in smart contracts. It does not only detect the vulnerabilities but also identifies the cause of vulnerabilities along with details. It also provides some recommendations related to particular security issues. It uses XPath queries to detect the patterns of vulnerabilities. An experiment was held to check the efficiency of SmartCheck by providing around 4.5k contracts and it was successful in detecting vulnerabilities in 99.9%.

Following are the descriptions of sections in this research paper. In Section II the literature review of some research papers is discussed. In Section III tools, section IV proposed architecture ,section V the methodology to implement smart security in smart contracts is described. The Section VI comparative analysis, VII conclusion & VIII contains bibliography..

II.RELATED WORK

This section involves reviews concentrated on smart contracts, its security and attacks which helps in investigating the security and vulnerabilities in smart contracts. Some reviews of related work are described as under:

The paper [1] presents some mistakes that are made in smart contracts and guide the people with some methods to prevent these mistakes. In this paper the students are targeted which are involved in the practices of smart contracts. The common mistakes include some programming, logical and security issues. The programming and logical errors commonly occurred in the encoding of complex state machines. The most common pitfall that occurs in smart contracts is the error that causes monetary leaks.

Smart contracts developed on blockchain have a huge impact on new businesses. It is a difficult task and started only in industrial and scientific fields. Wohrer et al. [2] presented security patterns with solutions based on grounded theory, applied on smart contract

data. These solutions are based on Solidity which is a popular programming language used in block chain. These security patterns can be applied to solve the security issues to avoid common attack scenarios.

Some of the vulnerabilities along with attacks in the architecture layer of smart contracts are discussed in paper [3]. Three aspects which are vulnerabilities, attacks and defense are presented in this paper. It presents a survey of Ethereum security systems. It presents 40 types of vulnerabilities in the architecture of Ethereum and also provides its root causes. Some defense techniques are discussed. It also highlights a factor that using a better programming language can help in making contract fault tolerant but cannot make the contract secure.

A new way for the transactions of cryptocurrency is smart contracts but there are a lot of vulnerabilities in this technique. An exploitable bug can cause a loss of a huge amount of money. A most common type of bug known as reentrancy bug is discussed in paper. This bug caused an attack known as DAO (Decentralized Autonomous Organization) in 2016, which resulted in a debt of $60 million worth of Ethereum cryptocurrency. Liu [4] presents a technique called *ReGuard* for the detection of bugs. It is a fuzzing-based technology that performs fuzz testing by generating random transactions. It identifies vulnerabilities and automatically flags them.

A framework known as Osiris, is presented in research [5]. This technique is designed to find bugs and errors in smart contracts. This tool works efficiently and detects a wider range of errors than the tools available. An experiment was performed to evaluate its performance, a large dataset is provided that contains around 1.2 million smart contracts. Smart contracts are being used in business and dealing with intellectual properties. But due to certain vulnerabilities in smart contracts many issues are reported which causes great financial losses. Many solutions have also been developed. After analyzing many researches Y Haung [6] addresses some vulnerabilities and like traditional software development lifecycle (SDLC) the contract lifecycle is also divided into four stages.

Checking the mechanism of transaction, seemed to require much attention as it is a critical function in smart contracts when it is dealing with live transfer of money. This makes it critical as to secure this requires different techniques. Some go for manual while some for automated prevention technique,while in **[7]** Jemin Andrew came up with idea of run time validation that found to be a strategic and efficient approach.However there is a downfall for this method which is overhead of runtime validation(excessive performance overhead),which can be costly for domains in blockchain. This issue can be integrated with Solythesis, tool of runtime validation for smart contracts(Ethereum), it uses as source compiler tool for solidity and analyzing/detect of error at compile time, however it also facilitates expressive or mechanism language(which uses quantifiers for allowing users to identify important functions of smart contracts)

In a research study, a method posted in **[9]**about checking of smart contract if is it exhibits vulnerability by using model checking and adopting rules of μ-Calculus taking care of vulnerabilities normally termed as Automata. It was thus evaluated on a dataset and predicting it through a precision and recall rate whether it is vulnerable or not. While other researchers gave their approach, one of the most maximax mechanisms that found to be useful is analyzing access control mechanisms in blockchain as it can lead to deadliest bugs that can be exposed to a greater harm for an entity. In **[8]** the paper, researcher think of this idea of using High

Granularity Metrics scheme,building white and blacklisting access control in an adaptive manner to filter out unwanted and unauthorized pickling with data.Thus we create many layers for learning, identifying, and alerting unusual behavior by increasing visibility through aggregated feature-level measurements.Another report by researchers shows in their paper is **[10]**,the introduction of tool Ethainter, a security analytics tool that cleans data from smart contracts to verify the information flow. Ethainter identifies fusion attacks that result in significant breaches by spreading contaminated information across numerous transactions. The analysis covers the entire blockchain, which consists of millions of accounts and hundreds of thousands of different smart contracts. We validate that Ethainter is more accurate than earlier methods for autonomous mining generation.

Researchers nowadays always look for a way to secure the transactions carried out in blockchain due to the critical nature.False positive is an issue usually addressed by many to use methods to overcome it but still it persists which If the problem is either not present or cannot be exploited. In one of the studies**[11]**,a formal approach part work is presented to address that issue. Our work builds upon our earlier analysis consisting of two stages: first, by considering the notion. Focus on the switch's function calls and LTL attributes to find the accuracy of the smart contract. These characteristics may be unique to the management or information flow of the contracts under verification. By suggesting formal characteristics ofLTL for vulnerabilities from the literature, we show that they can also be utilized to reveal flaws.In another study,**[12]** another tool HFContract-Fuzzer is introduced ,a Fuzzing-based approach that combines a go-fuzz-based Fuzzing engine with go-written smart contracts. We identified four of the five agreement vulnerabilities we found from well-known reassets using HFContractFuzzer, proving the efficacy of the suggested approach.

In the paper **[13]**, authors present a number of fresh security issues where a malicious party might influence the execution of a smart contract in order to profit. These flaws point to minute weaknesses in the platform's underlying distributed semantics knowledge. They suggest approaches to improve Ethereum's operational semantics in order to make contracts more secure. They created a symbolic execution tool called Oyente to help contract writers for the current Ethereum system detect possible security flaws. Oyente classifies 8, 833 of the 19, 366 currently in use Ethereum contracts as insecure, including the The DAO flaw that cost the company $60 million in June 2016. A number of studies with published code that are used in attacks made on Ethereum network are also taken into consideration.

Researches showed security flaws, faults, and vulnerabilities in the Ethereum smart contract. Applying the Self Destruct feature is the most effective way to terminate a contract at the blockchain device and transfer all of the Ethers included within the contract stability. As a result, when issues are discovered, many builders use this feature to cancel the agreement and set up a new one. A deep learning-based algorithm was used in one study **[14]** to retrieve the updated version of a destroyed contract in order to identify security flaws in Ethereum smart contracts. Then, using open card sorting, we look for security flaws in the upgraded versions. In another study **[15]** Authors personally found their defined contract flaws in 587 actual smart contracts by examining Feedback; they then made their dataset available to the public. Finally, they listed five effects brought on by contract flaws. These aid developers in comprehending the problems' symptoms and order of importance for elimination. Another study’s say **[16]** The authors explore the

topic of security of intelligent settlement programming in addition to providing an extensive taxonomy of all accepted security concerns and examining the safety code evaluation techniques utilized to identify acknowledged vulnerabilities. Through comparing their effectiveness and precision on acknowledged faults on a consultant pattern of weak contracts, they investigate Ethereum's safety code evaluation tools. To determine how comprehensive the most recent safety evaluation tools for Ethereum are, researchers looked at the performance of 4 safety tools—Oyente, Securify, Remix, and SmartCheck—as well as 21 stable and 24 tilted contracts.

The author gives **[17]** Their research sheds light on the smart-contract security field. Based on thorough research and excellent experiments we present an updated taxonomy of vulnerabilities, their architectural categorization, along with their severity level. They also examine some of the security technology to assess its accuracy, strength, and consistency. They have categorized the equipment in accordance with the manner their user interface and capacity for evaluation enable them to provide a "kingdom of the art" set of protection equipment on Ethereum. They create a matrix of protective gear and the vulnerabilities it covers in order to find gaps and missing vulnerability checks. In another work **[18]**, they provide a security assurance approach for smart contracts in order to prevent potential vulnerabilities while creating smart contracts. Their solution has two essential components. Developers can benefit from clearer understanding of their code structure with the topology diagram creation of invocation relationships for cross-file smart contracts. In addition, they broaden the range of logical dangers and may identify and pinpoint them using syntax analysis and symbolic execution. Together with syntax analysis, they can identify logic vulnerabilities to certain functions. They created the SASC tool based on these characteristics. It can offer Ethereum smart contracts with very strong quality assurance.

The authors of this research paper**[21]** define smart contracts as "a set of promises, specified in the digital form, including the agreement between the parties regarding these promises." The authors note that smart contracts can revolutionize the way that we do business by increasing efficiency and reducing the need for intermediaries. However, they also highlight a number of security and challenges associated with the use of smart contracts. One of the main security concerns surrounding smart contracts is their vulnerability to hacking. Since smart contracts are written in code, they can be subject to the same types of cyber attacks as any other computer system. This includes vulnerabilities such as code injection and replay attacks. In addition to security concerns, the authors also discuss a number of difficulties with implementation of smart contracts. These include issues with interoperability between different smart contract platforms, the lack of legal recognition for smart contracts in some jurisdictions, and the need for standardization in order to facilitate widespread adoption. Overall, the authors conclude that while smart contracts can revolutionize the way that we do business, there are also a number of security and challenges that must be addressed in order for them to reach their full potential. Further research is needed to address these issues and ensure the secure and successful deployment of smart contracts.

In this paper **[23]**, the authors present that blockchain technology enables secure and tamper-proof communication and data storage on a global scale. Furthermore, this technology is being used in smart contracts which are small pieces of programs that facilitate agreements between parties. The authors, however, point out that there are security weaknesses

in Ethereum blockchain-based smart contracts that can lead to substantial financial losses. Henceforth, the author has carried out an in-depth examination of these vulnerabilities, examining the detection tools, actual attacks, and measures to prevent them, as well as making comparisons between them. The authors also discuss the challenges and issues with Ethereum-based blockchain smart contracts. In conclusion, the author suggests that as more features are added to Ethereum smart contracts over time, this may result in an increase in potential security vulnerabilities. This will require enhancing existing detection tools as well as developing new detection tools. For finding the new vulnerabilities, further research would also be required as overtime new security and scalability issues will increase.

The authors of this paper **[24]** conducted research utilizing both qualitative and quantitative methods, including interviews with 13 participants and surveys with 156 respondents from 35 countries across six continents, to examine the opinions and practices of professionals regarding the security of smart contracts. Furthermore, they examined the factors that motivate or discourage smart contract security among practitioners and how security measures are integrated into the development process. Besides providing useful instruction on utilizing code effectively, the impacts of tools, and proactive measures to improve smart contract security, the authors also observed several distinctions between smart contract security and regular security. Additionally, the research found that the type of blockchain platform utilized greatly impacts the perceived and actual security of smart contract development. From these conclusions, the study's authors have proposed potential avenues for further study and provided suggestions for those implementing smart contracts.

The authors of this paper **[25]** argue that the popularity of smart contracts is due to their ability to self-enforce without the need for a trusted third party. However, it has been discovered that a significant number of these contracts have weaknesses in their structure that permit attackers to take valuable possessions from the individuals involved. These vulnerabilities are a result of poor design and highlight the importance of carefully designing and implementing smart contracts to prevent exploitation. For this purpose, the authors have proposed a debt-conscious method to evaluate the vulnerabilities in the design of smart contracts. This approach involves identifying any weaknesses through security analysis techniques and then determining the impact of these vulnerabilities using the technical debt concept, including the principal and interest. Through this method, the identification of issues related to security design was enhanced and developers were able to prioritize and tackle vulnerabilities in smart contracts through evaluating the impact on technical debt.

III.TOOLS AND TECHNOLOGIES

Smart contracts: Unsurprisingly, a smart contract is a set of files executed on the blockchain that is linked to a specific environment. These are frequently employed to automate the merger execution process. This is because, barring the involvement of certain middlemen and the absence of events, all stockholders can now be confident in the outcome. While the environments are connected, workflows can be automated and additional procedures can be started.

ERC-20 Token: Numerous cryptocurrencies have been developed in the modern era. However, the majority of them are accomplished via Ethereum-based smart contracts, often referred to as tokens, which also hold their own blockchain

configuration. Ethereum offers a variety of memorial flags that reflect the opinions of related coins. In this scenario, numerous tokens can interact properly and be reused by diverse users (like, wallets and exchange markets).

Remix: All skill levels of users can utilise the well-liked smart contract creation tool Remix IDE. It is simple to use, requires no setup, has a quick development cycle, a large selection of plugins, and a user-friendly GUI. There are several forms available for it, including a desktop package, an internet application, a VS Code plugin, and more.

SmartCheck: The result of a thorough investigation to identify more code explosions and problems in Ethereum Smart Contracts registered under the Solidity installation term is known as SmartCheck.

Securify: On their website, Securify describes itself as a net-positioned security platform with industrialized components (to enable everyone to check smart contracts), assurance (for precise risk), and flexibility. I promise (for some of these). days to seize a particular exposure). Securify uses unambiguous evidence but also relies on tests of inflexible reasoning. Project reordering, repeated calls, insecure codified styles, unexpected heavenly floods, and the usage of inaccurate recommendations are the main difficulties it addresses. Repeated calls, unexpected downpours of rain, and one of the related codified style checks are warranted, nevertheless (sufficient approach requirement).

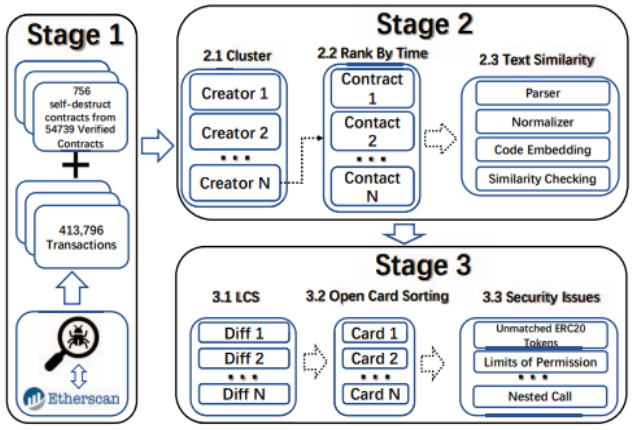
F* Framework:F* is a framework created by Microsoft Research that enables the assessment of the functional correctness and runtime security of Ethereum smart contracts. It employs formal verification to uncover possible vulnerabilities like re-entry and exception issues by translating Solidity or bytecode to F* (Functional Programming Language). The security of smart contracts may be ensured with the help of this architecture.

Oyente: Oyente uses symbolic execution to identify possible security flaws, such as reentrancy, timestamp reliance, mishandled exceptions, and dependence on transaction sequencing. The tool has the ability to examine both Solidity and a smart contract's bytecode. It could only be used via a command line interface in its early stages. It currently offers a web-based interface that is more user-friendly.

IV.PROPOSED ARCHITECTURE

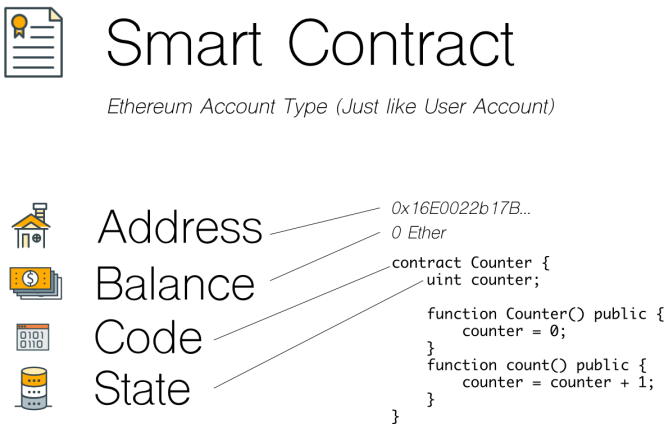
A. Architecture Design:

Architecture overview for comparing older versions to identify security flaws in Ethereum smart contracts.



B. Smart Contract Design:

Anyone can use a smart contract's functionality after it has been installed on the Ethereum network. The feature can have security measures that prevent users from utilizing it.



V.METHODOLOGIES

A)Unencrypted Private Data On-Chain: Vulnerability

Security is an issue when a Critical Private Variable may be accessed via a Public Method. Contrary to popular belief, private variables can be read. Even if the contract is not disclosed, attackers can nevertheless study contract transactions to find values kept in the contract state. To guard against this issue, it is essential to avoid keeping private data that has not been encrypted in contract code or state.

odd_even.sol

```sol
pragma solidity ^0.5.0;

contract OddEven {
    struct Player {
        address addr;
        uint number;
    }

    Player[2] private players;
    uint count = 0;

    function play(uint number) public payable {
            require(msg.value == 1 ether, 'msg.value must be 1 eth');
            players[count] = Player(msg.sender, number);
            count++;
            if (count == 2) selectWinner();
    }

    function selectWinner() private {
            uint n = players[0].number + players[1].number;
            (bool success, ) = players[n%2].addr.call.value(address(this).balance)("");
            require(success, 'transfer failed');
            delete players;
            count = 0;
    }
}
```

Prevention

An intriguing way to simulate a secret store of information is to utilise a commit to disclose a pattern. Users initially provide the secret information's hash using this way. Each user then exposes their vote, which can then be back-verified by the other users, when everyone has provided their information. To guarantee the privacy of the stored information, several different variations of this strategy have been tried.

odd_even_fixed.sol

In this file on line 35 We verify the hash to demonstrate the player's integrity.

```sol
pragma solidity ^0.5.0;

contract OddEven {
    enum Stage {
        FirstCommit,
        SecondCommit,
        FirstReveal,
        SecondReveal,
        Distribution
    }

    struct Player {
        address addr;
        bytes32 commitment;
        uint number;
    }
    Player[2] private players;
    Stage public stage = Stage.FirstCommit;
    function play(bytes32 commitment) public payable {
        uint playerIndex;
        if(stage == Stage.FirstCommit) playerIndex = 0;
        else if(stage == Stage.SecondCommit) playerIndex = 1;
        else revert("only two players allowed");
        require(msg.value == 2 ether, 'msg.value must be 2 eth');
        players[playerIndex] = Player(msg.sender, commitment, 0);
        if(stage == Stage.FirstCommit) stage = Stage.SecondCommit;
        else stage = Stage.FirstReveal;
    }
    function reveal(uint number, bytes32 blindingFactor) public {
        require(stage == Stage.FirstReveal || stage == Stage.SecondReveal, "wrong stage");
        uint playerIndex;
        if(players[0].addr == msg.sender) playerIndex = 0;
        else if(players[1].addr == msg.sender) playerIndex = 1;
        else revert("unknown player");
        require(keccak256(abi.encodePacked(msg.sender, number, blindingFactor)) == players[playerIndex].commitment, "invalid hash");
        players[playerIndex].number = number;
        if(stage == Stage.FirstReveal) stage = Stage.SecondReveal;
        else stage = Stage.Distribution;
    }
    function distribute() public {
        require(stage == Stage.Distribution, "wrong stage");
        uint n = players[0].number + players[1].number;
        players[n%2].addr.call.value(3 ether)("");
        players[(n+1)%2].addr.call.value(1 ether)("");
        delete players;
        stage = Stage.FirstCommit;
    }
}
```

B)Hash Collisions With Multiple Variable Length Arguments: Vulnerability

When using abi.encodePacked() with several variable length parameters, a security flaw called Authentication Bypass through Capture-Replay may appear. This is due to the possibility of a hash collision under certain circumstances. The issue is that, regardless of whether an element is a part of an array, abi.encodePacked() packs all items in sequence. As long as all of the components are in the same sequence, an attacker can transfer elements across arrays and it will still return the same encoding. An attacker might take advantage of this weakness in a signature verification scenario by changing the order of items in a prior function call to successfully evade permission. When using abi.encodePacked() and other comparable methods, special consideration must be given to the order of the components in order to stop this attack.

access_control.sol

access_control.sol

```sol
pragma solidity ^0.5.0;

import "./ECDSA.sol";
contract AccessControl {
    using ECDSA for bytes32;
    mapping(address => bool) isAdmin;
    mapping(address => bool) isRegularUser;
    function addUsers(
        address[] calldata admins,
        address[] calldata regularUsers,
        bytes calldata signature
    )
        external
    {
        if (!isAdmin[msg.sender]) {
            bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
            address signer = hash.toEthSignedMessageHash().recover(signature);
            require(isAdmin[signer], "Only admins can add users.");
        }
        for (uint256 i = 0; i < admins.length; i++) {
            isAdmin[admins[i]] = true;
        }
        for (uint256 i = 0; i < regularUsers.length; i++) {
            isRegularUser[regularUsers[i]] = true;
        }
    }
}
```

Prevention

A matching signature cannot be achieved with varied arguments when using abi.encodePacked(). It's crucial to utilize fixed length arrays or to limit user access to the parameters used by abi.encodePacked(). Use of abi.encode(), which is less vulnerable to this flaw, is an alternate option. When using abi.encodePacked, it is essential to provide the necessary safeguards to avoid this kind of vulnerability ().

access_control_fixed.sol

access_control.sol | access_control_fixed.sol

```sol
pragma solidity ^0.5.0;

import "./ECDSA.sol";

contract AccessControl {
    using ECDSA for bytes32;
    mapping(address => bool) isAdmin;
    mapping(address => bool) isRegularUser;
    function addUser(
        address user,
        bool admin,
        bytes calldata signature
    )
        external
    {
        if (!isAdmin[msg.sender]) {
            bytes32 hash = keccak256(abi.encodePacked(user));
            address signer = hash.toEthSignedMessageHash().recover(signature);
            require(isAdmin[signer], "Only admins can add users.");
        }
        if (admin) {
            isAdmin[user] = true;
        } else {
            isRegularUser[user] = true;
        }
    }
}
```

C) Function Default Visibility: Vulnerability

Security flaws can result from improper coding standards compliance, such as when functions don't have a declared visibility type and instead default to public. If a developer forgets to correctly establish the visibility, this may lead to illegal or unintentional state changes. Developers must follow coding guidelines and make sure that all functions have the correct visibility type configured in order to avoid this kind of vulnerability.

visibility_not_set.sol

visibility_not_set.sol

```
pragma solidity ^0.4.24;

contract HashForEther {
    function withdrawWinnings() {
        require(uint32(msg.sender) == 0);
        _sendWinnings();
    }

    function _sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

Prevention

Different visibility categories, including external, public, internal, and private, are available for functions. For each function, it's crucial to select the proper visibility type. This can significantly lessen a contract system's vulnerability and make it more resistant to assaults. It is possible to significantly increase the security of the contract system by paying attention to function visibility.

visibility_not_set_fixed.sol

visibility_not_set.sol | visibility_not_set_fixed.sol

```
pragma solidity ^0.4.24;

contract HashForEther {
    function withdrawWinnings() public {
        require(uint32(msg.sender) == 0);
        _sendWinnings();
    }
    function _sendWinnings() internal{
        msg.sender.transfer(this.balance);
    }
}
```

D)Write to Arbitrary Storage Location:

Vulnerability

It is related to the security term Write-what-where Condition A blockchain-based piece of code known as a "smart contract" is intended to facilitate,verify, and enforce the negotiation or performance of a contract. Smart contracts contain data, such as the owner of the contract, which is stored in a specific location on the blockchain known as the Ethereum Virtual Machine (EVM). The smart contract ensures that only authorized users or other contracts are able to make changes to this data. If an attacker is able to bypass these authorization checks and write to the contract's data storage, they could potentially corrupt the stored information. For example, an attacker might overwrite the field that stores the address of the contract owner, potentially allowing the attacker to take control of the contract.

It's important for smart contracts to have secure authorization checks to prevent attackers from tampering with the stored data. If an attacker is able to modify the data stored in a smart contract, they could potentially gain unauthorized access to sensitive information or take control of the contract. This could lead to significant problems, such as financial loss or damage to reputation. Therefore, it's important to carefully design and implement smart contracts to ensure that they are secure and resistant to attacks.

arbitrary_location_write_simple.sol

```
1  pragma solidity ^0.4.25;
2
3  contract Wallet {
4      uint[] private bonusCodes;
5      address private owner;
6
7      constructor() public {
8          bonusCodes = new uint[](0);
9          owner = msg.sender;
10     }
11
12     function () public payable {
13     }
14
15     function PushBonusCode(uint c) public {
16         bonusCodes.push(c);
17     }
18
19     function PopBonusCode() public {
20         require(0 <= bonusCodes.length);
21         bonusCodes.length--;
22     }
23
24     function UpdateBonusCodeAt(uint idx, uint c) public {
25         require(idx < bonusCodes.length);
26         bonusCodes[idx] = c;
27     }
28
29     function Destroy() public {
30         require(msg.sender == owner);
31         selfdestruct(msg.sender);
32     }
33 }
```

Fig. arbitrary_location.sol contract code

Prevention

It is advice to ensure that data structures within the contract do not overwrite each other. If one data structure is written to, it could potentially overwrite another data structure if they share the same storage space. This could lead to data loss or corruption, and could potentially compromise the security and integrity of the contract. To prevent this, it is important to ensure that writes to one data structure cannot inadvertently overwrite entries of another data structure. This may involve careful design and implementation of the contract to ensure that data structures are stored in separate areas of the storage space, or by implementing measures to protect against accidental overwriting.

Arbitrary_location_write_simple_fixed.sol

In line 25 Since you now have to push very many codes this is no longer an arbitrary write.

```
1  pragma solidity ^0.4.25;
2
3  contract Wallet {
4      uint[] private bonusCodes;
5      address private owner;
6
7      constructor() public {
8          bonusCodes = new uint[](0);
9          owner = msg.sender;
10     }
11
12     function () public payable {
13     }
14
15     function PushBonusCode(uint c) public {
16         bonusCodes.push(c);
17     }
18
19     function PopBonusCode() public {
20         require(0 < bonusCodes.length);
21         bonusCodes.length--;
22     }
23
24     function UpdateBonusCodeAt(uint idx, uint c) public {
25         require(idx < bonusCodes.length);
26         bonusCodes[idx] = c;
27     }
28
29     function Destroy() public {
30         require(msg.sender == owner);
31         selfdestruct(msg.sender);
32     }
33 }
34
```

E) Weak Sources of Randomness from Chain Attributes:

Vulnerability

It is related to the security term Use of Insufficiently Random Values Pseudo-random number generators (PRNGs) are commonly used in applications, such as gambling DApps, where a source of randomness is needed. However, generating a strong source of randomness on the Ethereum blockchain can be difficult. This is because the Ethereum blockchain is decentralized, meaning that it is maintained by a network of computers, rather than a central authority. As a result, it can be difficult to ensure that the source of randomness is truly random, as it may be influenced by the actions of the miners who maintain the network.

One common method of generating randomness on the Ethereum blockchain is to use the block.timestamp field, which records the time at which a block was added to the blockchain. The problem with this approach is that a miner can

choose to submit any timestamp that is a few seconds or less away from the present time and the network will still accept their block.Other methods, such as using the blockhash or block.difficulty fields, are also insecure, as they are controlled by the miner. In cases where the stakes are high, a miner may be motivated to manipulate the source of randomness in order to increase their chances of winning. This could be done, for example, in short period of time mining the large number of blocks and selecting the block that has the required block hash to ensure a win, while discarding all other blocks.

Guess_the_random_number.sol

```
pragma solidity ^0.4.21;

contract GuessTheRandomNumberChallenge {
    uint8 answer;

    function GuessTheRandomNumberChallenge() public payable {
        require(msg.value == 1 ether);
        answer = uint8(keccak256(block.blockhash(block.number - 1), now));
    }

    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }

    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);

        if (n == answer) {
            msg.sender.transfer(2 ether);
        }
    }
}
```

Prevention

There are several methods that can be used to generate randomness on the Ethereum blockchain in a more secure way. One method is to use a commitment scheme, such as RANDAO, which allows multiple parties to commit to a random value without revealing it until a later time. This can help to ensure that the value is truly random, as it is not controlled by any single party.

Another method is to use external sources of randomness via oracles, such as Oraclize. Oracles are external entities that provide data to smart contracts from off-chain sources. By using multiple oracles, it is possible to increase the security of the randomness generation process, as the risk of any single oracle being compromised is reduced. However, The use of this method necessitates trust in the oracles, which must be noted.so it is important to carefully consider the reputation and reliability of the oracles being used.

A third method is to use Bitcoin block hashes, as they are more expensive to mine and may be less likely to be manipulated by a single miner. However, this method requires a secure way to access the Bitcoin blockchain from within the Ethereum blockchain, which may be challenging to implement

guess_the_random_number_fixed.sol

```
pragma solidity ^0.4.25;

contract GuessTheRandomNumberChallenge {
    uint8 answer;
    uint8 commitedGuess;
    uint commitBlock;
    address guesser;

    function GuessTheRandomNumberChallenge() public payable {
        require(msg.value == 1 ether);
    }

    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }


    function guess(uint8 _guess) public payable {
        require(msg.value == 1 ether);
        commitedGuess = _guess;
        commitBlock = block.number;
        guesser = msg.sender;
    }
    function recover() public {
      require(block.number > commitBlock + 20 && commitBlock+20 > block.number - 256);
      require(guesser == msg.sender);

      if(uint(blockhash(commitBlock+20)) == commitedGuess){
        msg.sender.transfer(2 ether);
      }
    }
}
```

.

F) Signature Malleability: Vulnerability

It is related to the security term Improper Verification of Cryptographic SignatureCryptographic signature systems are commonly used in Ethereum contracts to provide security and authenticity for transactions and other operations. A signature is a piece of data that is created using a private key and is designed to be unique to the owner of the private key. However, it is

possible for signatures to be altered without the possession of the private key and still be considered valid. This is because the Ethereum Virtual Machine (EVM) includes several "precompiled" contracts, including ecrecover, which is designed to execute the elliptic curve public key recovery process.

A malicious user can use this precompiled contract to slightly modify the values of v, r, and s in a signature, creating other valid signatures. This could potentially allow an attacker to replay previously signed messages or perform other types of attacks. If a system on the contract level relies on signature verification, it may be susceptible to these types of attacks if the signature is part of the signed message hash. To protect against these types of attacks, it is important to carefully design and implement cryptographic signature systems in Ethereum contracts, and to consider the potential risks associated with signature modification.

Transaction_malleablity.sol

```
pragma solidity ^0.4.24;

contract transaction_malleablity{
  mapping(address => uint256) balances;
  mapping(bytes32 => bool) signatureUsed;

  constructor(address[] owners, uint[] init){
    require(owners.length == init.length);
    for(uint i=0; i < owners.length; i ++){
      balances[owners[i]] = init[i];
    }
  }

  function transfer(
       bytes _signature,
       address _to,
       uint256 _value,
       uint256 _gasPrice,
       uint256 _nonce)
    public
    returns (bool)
    {
      bytes32 txid = keccak256(abi.encodePacked(getTransferHash(_to, _value, _gasPrice, _nonce), _signatur
      require(!signatureUsed[txid]);

      address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);

      require(balances[from] > _value);
      balances[from] -= _value;
      balances[_to] += _value;

      signatureUsed[txid] = true;
    }

    function recoverTransferPreSigned(
        bytes _sig,
        address _to,
        uint256 _value,
        uint256 _gasPrice,
```

```
function getTransferHash(
    address _to,
    uint256 _value,
    uint256 _gasPrice,
    uint256 _nonce)
  public
  view
returns (bytes32 txHash) {
    return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasPrice, _nonce);
}
function getSignHash(bytes32 _hash)
  public
  pure
returns (bytes32 signHash)
{
    return keccak256("\x19Ethereum Signed Message:\n32", _hash);
}
function ecrecoverFromSig(bytes32 hash, bytes sig)
  public
  pure
returns (address recoveredAddress)
{
    bytes32 r;
    bytes32 s;
    uint8 v;
    if (sig.length != 65) return address(0);
    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
    if (v < 27) {
      v += 27;
    }
    if (v != 27 && v != 28) return address(0);
    return ecrecover(hash, v, r, s);
}
```

Prevention

It is generally not recommended to include a signature in the signed message hash when performing signature verification on an Ethereum contract. This is because, as mentioned in the previous response, signatures can be altered without the possession of the private key and still be considered valid. If the signature is included in the signed message hash, an attacker could potentially create valid signatures that would allow them to replay previously signed messages or perform other types of attacks.

To protect against these types of attacks, it is important to ensure that signature verification is performed in a way that does not rely on the inclusion of the signature in the signed message hash. This may involve using other methods of verifying the authenticity of the signature, such as checking the public key associated with the signature or using additional data in the signed message. By taking these precautions, it is possible to reduce the risk of attacks on the contract and ensure that it is secure and reliable.

Transaction_malleablity_fixed.sol

```solidity
pragma solidity ^0.4.24;

contract transaction_malleablity{
  mapping(address => uint256) balances;
  mapping(bytes32 => bool) signatureUsed;

  constructor(address[] owners, uint[] init){
    require(owners.length == init.length);
    for(uint i=0; i < owners.length; i ++){
      balances[owners[i]] = init[i];
    }
  }

  function transfer(
      bytes _signature,
      address _to,
      uint256 _value,
      uint256 _gasPrice,
      uint256 _nonce)
    public
  returns (bool)
  {
    bytes32 txid = getTransferHash(_to, _value, _gasPrice, _nonce);
    require(!signatureUsed[txid]);

    address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);

    require(balances[from] > _value);
    balances[from] -= _value;
    balances[_to] += _value;

    signatureUsed[txid] = true;
  }

  function recoverTransferPreSigned(
      bytes _sig,
      address _to,
      uint256 _value,
      uint256 _gasPrice,
      uint256 _nonce)
    public
    view
  returns (address recovered)
  {
      return ecrecoverFromSig(getSignHash(getTransferHash(_to, _value, _gasPrice, _nonce)), _sig);
  }

  function getTransferHash(
      address _to,
      uint256 _value,
      uint256 _gasPrice,
      uint256 _nonce)
    public
    view
  returns (bytes32 txHash) {
      return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasPrice, _nonce);
  }

  function getSignHash(bytes32 _hash)
    public
    pure
  returns (bytes32 signHash)
  {
      return keccak256("\x19Ethereum Signed Message:\n32", _hash);
  }

  function ecrecoverFromSig(bytes32 hash, bytes sig)
    public
    pure
  returns (address recoveredAddress)
  {
      bytes32 r;
      bytes32 s;
      uint8 v;
      if (sig.length != 65) return address(0);
      assembly {
          r := mload(add(sig, 32))
          s := mload(add(sig, 64))
          v := byte(0, mload(add(sig, 96)))
      }
      if (v < 27) {
        v += 27;
      }
      if (v != 27 && v != 28) return address(0);
      return ecrecover(hash, v, r, s);
    }
}
```

G) Shadowing State Variables: Vulnerability

It is related to the security term Adherence to Coding Standards The Ethereum blockchain's smart contracts are created using the computer language Solidity. One potential issue that can arise when using Solidity is ambiguous naming of state variables, which occurs when a contract inherits from another contract that defines state variables with the same name. In this case, there will be two separate versions of the variable, one that is accessed from contract A and another that is accessed from contract B. This can lead to confusion and potentially cause security issues if the ambiguity is not noticed.

Shadowing state variables can also occur within a single contract when there are multiple definitions of the same variable at the contract and function level. This can also lead to confusion and potentially cause issues with the correct functioning of the contract. To avoid these issues, it is important to carefully design and structure smart contracts to ensure that state variables are named and defined in a clear and consistent manner. This can help to reduce the risk of ambiguity and guarantee the trustworthiness and security of smart contracts,

TokenSale.sol

```solidity
pragma solidity 0.4.24;

contract Tokensale {
    uint hardcap = 10000 ether;

    function Tokensale() {}

    function fetchCap() public constant returns(uint) {
        return hardcap;
    }
}

contract Presale is Tokensale {
    uint hardcap = 1000 ether;

    function Presale() Tokensale() {}
}
```

Prevention

To guarantee the trustworthiness and security of smart contracts, it is important to carefully review the storage variable layouts for contract systems and remove any ambiguities that may arise. This can help to prevent issues such as shadowing and ambiguous

naming of state variables, which can lead to confusion and potentially cause problems with the correct functioning of the contract

One way to identify potential issues with storage variable layouts is to check for warnings, as these can flag problems within a single contract. By paying attention to compiler warnings and addressing any issues that they may highlight, it is possible to reduce the risk of security vulnerabilities and improve the overall quality of the contract.

TokenSale_fixed.sol

```
pragma solidity 0.4.25;


contract Tokensale {
    uint public hardcap = 10000 ether;

    function Tokensale() {}

    function fetchCap() public constant returns(uint) {
        return hardcap;
    }
}

contract Presale is Tokensale {
    function Presale() Tokensale() {
        hardcap = 1000 ether;
    }
}
```

VI.COMPARATIVE ANALYSIS

Vulnerability:

Analysis between different vulnerabilities showed as in the figure, which shows the impact, severity score and distribution growing percentage found normally in codebases in smart-contracts

Contract Defect	Distribution	Score	#Defects	Impac
Incorrect Constructor Name		4.50	25 (4.26%)	IP3
Insufficient Gas Griefing		4.28	5 (0.85%)	IP2
Message call with hardcoded gas amount		4.54	5 (0.85%)	IP1
Reentrancy		4.10	84 (14.31%)	IP3
Arithmetic Over/Underflows		4.45	13 (2.21%)	IP2

Fig. Analysis of vulnerabilities

Smart Contracts:

Occurrence of vulnerabilities between unique and normal contracts as depicted in the figure

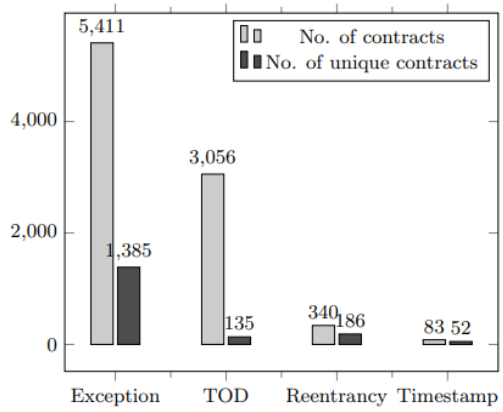


Fig.The number of contracts affected by each security issue.

Security Tools:

We carried out a thorough investigation to find smart contract systems' known weaknesses. We conducted a literature analysis, examined online sources, and kept an eye on discussions on smart contract security issues in relevant web pages, blogs, and Reddit forums. For further information, we also contacted creators and users of particular security solutions via email and group conversations (such Slack channels and Gitter). We chose Remix as our development platform since it is thought to be more safe than the alternatives based on the data we obtained.

Security Tool	ReEntrancy	Timestamp dependency	TOD[26]	Mishandled exceptions	Immutable Bugs	tx.orgin usage	Gas costly patterns	Blockhash usage
Oyente	✓	✓	✓	✓	✓	X	X	X
Remix	✓	✓	X	✓	X	✓	✓	✓
F*	✓	X	X	✓	X	X	X	X
Gasper	X	X	X	X	X	X	✓	X
Securify	✓	X	✓	✓	X	✓	X	X
S. Analysis	X	X	X	✓	X	X	X	X
SmartCheck	✓	✓	✓	✓	✓	✓	✓	X
Imandra	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Mythril	✓	X	X	✓	✓	✓	X	X

Fig. Tools / Vulnerabilities Matrix

VII.CONCLUSION

This paper highlights the improvement in the field of blockchain, the havocs with different challenges and openings that are related with these innovations and their straightforward solutions. It is a shared transaction ledger that contains records of exchanges through consensus-based algorithms. Its scope was restricted fairly as there were few transactions but in today's world its scope expanded through utilization of smart contracts in order to computerize different assignments by decreasing their trouble and making them more productive. Blockchain is incorporated with different nodes having a consensus system whereas minings is done for assurity of data consistency and authenticity. There are still some challenges that need to be covered which includes security breach, scalability, increasing market cap size. Some other problems include which programming languages to be used for security maintenance.Making sure of the security of smart contracts is still a top priority of the researchers because the dealings of transactions made our loss unbearable. Thus, we put about some secure coding methods in this paper to assure the security of these contracts making it safe from hijacking transactions. Still there is a room for vulnerability occurrence as development is still undergoing in this giant technology.

VIII. BIBLIOGRAPHIES

Kabeer Ahmed is an undergraduate student at NED University of Engineering and Technology in Software Engineering. His main areas of interest are Offensive Security like OS Exploitation, Web Exploitation and Defensive Security like Digital Forensics, Cryptography. He has hands-on experience in Data Extraction and web penetration testing. He is keenly interested in making this career in this Cyber Security field.

Abdul Moiz is a student at NED University of Engineering and Technology in Karachi, Pakistan. He is currently pursuing a degree in software engineering with a focus on web development stack MERN and web 3.0 technologies. His interests include the latest advancements in web development, blockchain technology and its application in web3.0. He is an active learner and always keen to explore new opportunities to enhance his skills.

Fizza Mariyam is an undergraduate student studying Software Engineering at NED University of

Engineering and Technology. She is determined and enthusiastic, and uses her skills in her interested domains like Blockchain, Artificial Intelligence, and Quality Assurance to contribute to making a positive impact on the software industry. She also has a strong interest in research.

REFERENCES

[1] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, “Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab,” in Proc. 20th Int. Conf. Financial Cryptography Data Security, 2016, pp. 79–94.

[2] M. Wohrer and U. Zdun, “Smart contracts: Security patterns in the Ethereum ecosystem and Solidity,” in Proc. Int. Workshop Blockchain Oriented Softw. Eng., 2018, pp. 2–8

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, ``A survey on Ethereum systems security," *ACMComput. Surv.*, vol. 53, no. 3, pp. 1_43, Jul. 2020, doi: 10.1145/3391195.

[4] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, ``ReGuard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. Companion*, May 2018, pp. 65_68.

[5] C. F. Torres, J. Schütte, and R. State, ``Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 19.

[6] Y. Huang, Y. Bian, R. Li, J. L. Zhao and P. Shi, "Smart Contract Security: A Software Lifecycle Perspective," in IEEE Access, vol. 7, pp. 150184-150202, 2019, doi: 10.1109/ACCESS.2019.2946988.

[7] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contracts with runtime validation. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 438–453. https://doi.org/10.1145/3385412.3385982

[8]Venkata Siva Vijayendra Bhamidipati, Michael Chan, Derek Chamorro, Arpit Jain, and Ashok Murthy. 2019. Adaptive Security for Smart Contracts using High Granularity Metrics. Proceedings of the 3rd International Conference on Vision, Image and Signal Processing. Association for Computing Machinery, New York, NY, USA, Article 83, 1–6. https://doi.org/10.1145/3387168.3387214

[9]Giuseppe Crincoli, Giacomo Iadarola, Piera Elena La Rocca, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. 2022. Vulnerable Smart Contract Detection by Means of Model Checking. In Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI '22). Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/3494106.3528672

[10]Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 454–469. https://doi.org/10.1145/3385412.3385990

[11]Ikram Garfatta, Kaïs Klai, Mohamed Graïet, and Walid Gaaloul. 2022. Model checking of vulnerabilities in smart contracts: a solidity-to-CPN approach. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22). Association for Computing Machinery, New York, NY, USA, 316–325. https://doi.org/10.1145/3477314.3507309

[12]Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection. In Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 321–328. https://doi.org/10.1145/3463274.3463351

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269.

[14] J. Chen, "Finding Ethereum Smart Contracts Security Issues by Comparing History Versions," 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 1382-1384.

[15] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, "Defining Smart Contract Defects on Ethereum," in IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 327-345, 1 Jan. 2022, doi: 10.1109/TSE.2020.2989002.

[16] R. Sujeetha and C. A. S. Deiva Preetha, "A Literature Survey on Smart Contract Testing and Analysis for Smart Contract Based Blockchain Application Development," 2021 2nd International Conference on Smart Electronics and Communication (ICOSEC), 2021, pp. 378-385, doi: 10.1109/ICOSEC51865.2021.9591750.

[17] A. Dika and M. Nowostawski, "Security Vulnerabilities in Ethereum Smart Contracts," 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2018, pp. 955-962, doi: 10.1109/Cybermatics_2018.2018.00182.

[18] E. Zhou et al., "Security Assurance for Smart Contract," 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2018, pp. 1-5, doi: 10.1109/NTMS.2018.8328743.

[19]Cryptopedia Staff , “What Was The DAO?” , Cryptopedia, Powered by Gemini, https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-dao-hack

[20] Singh, Amritraj, et al. "Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities." Computers & Security 88 (2020): 101654.

[21] Amin, Md. Ratul & Zuhairi, Megat & Saadat,. (2020). A Survey of Smart Contracts: Security and Challenges. International Journal of Advanced Science and Technology. 9867-9878.

[22] Tang, Song, et al. "Blockchain-Enabled Social Security Services Using Smart Contracts." IEEE Access 10 (2022): 73857-73870

[23] Kushwaha, Satpal Singh, et al. "Systematic review of security vulnerabilities in ethereum blockchain smart contract." IEEE Access (2022).

[24] Wan, Zhiyuan, et al. "Smart contract security: A practitioners' perspective." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.

[25] Ahmadjee, Sabreen, Carlos Mera-Gómez, and Rami Bahsoon. "Assessing smart contracts security technical debts." 2021 IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE, 2021.