EAI Endorsed Transactions
on Internet of Things
Research Article EAI.EU

Major vulnerabilities in Ethereum smart contracts: Investigation and statistical analysis

Mohammad Pishdar[1], Mahdi Bahaghighat[2,*], Rajeev Kumar[3], Qin Xin[4]

[1]Computer Engineering Department, Bu-Ali Sina University, Hamedan, Iran
[2]Computer Engineering Department, Imam Khomeini International University, Qazvin, Iran
[3]Department of Computer Science and Engineering, Delhi Technological University, Delhi, India
[4]Faculty of Science and Technology, University of the Faroe Islands, Faroe Islands

Abstract

The general public is becoming increasingly familiar with blockchain technology. Numerous new applications are made possible by this technology's unique features, which include transparency, strong security via cryptography, and distribution. These applications need certain programming tools and interfaces to be implemented. This is made feasible by smart contracts. If the prerequisites are satisfied, smart contracts are carried out automatically. Any mistake in smart contract coding, particularly security-related ones, might have an impact on the project as a whole, available funds, and important data. The current paper discusses the flaws of the Ethereum smart contract in this respect. By examining publically accessible scientific sources, this work aims to present thorough information about vulnerabilities, examples, and current security solutions. Additionally, a substantial collection of current Ethereum (ETH) smart contracts has undergone a static code examination to conduct the vulnerability-finding procedure. The output has undergone assessments and statistical analysis. The study's conclusions demonstrate that smart contracts have several distinct flaws, including arithmetic flaws, that developers should be more aware of. These vulnerabilities and the solutions that can be used to address them are also included.

Keywords: Blockchain Security, Smart Contract Security, Cryptocurrency Security, Smart contracts Attacks

Received on 14 02 2024, accepted on 17 12 2024, published on 18 12 2024

Copyright © 2024 M. Pishdar *et al.*, licensed to EAI. This is an open access article distributed under the terms of the CC BY-NC-SA 4.0, which permits copying, redistributing, remixing, transformation, and building upon the material in any medium so long as the original work is properly cited.

doi: 10.4108/eetiot.5120

*Corresponding author. Email: bahaghighat@eng.ikiu.ac.ir

1. Introduction

Blockchain is a kind of distributed database that combines data replication with chain-based cryptography to offer distribution, high security (because of encryption and increased availability), and transaction transparency. Blockchain technology has gained popularity in recent years among programmers, IT firms, and even users of virtual worlds as a leading solution for building distributed databases and employing collective agreements. Multiple networks have been made available to the general public as a blockchain platform following the technology competitions and the presentation of various blockchain solutions in line with the better development of applications. We can list some of the most well-known of these networks as Bitcoin, Ethereum, Solana, and Avalanche. To offer their users better services, each blockchain has published a variety of protocols in this area. We might mention different consensus systems between the nodes of a distributed network of information as examples of these protocols [1-3].

Several blockchain projects have created a middle and high-level layer as a user interface for developers after developing the necessary hardware platform and communication protocols. This work will make it easier for blockchain network developers to create a wide range of applications and handle the intricate technical and specialized requirements of the blockchain's foundational layer. This platform is known as smart contracts [1-2]. Programmers can use data structures, functions, and different object-oriented systems in smart contracts, as well as a number of unique functions and libraries that are accessible over the blockchain network. In other words, they

may quickly record information or query it on the infrastructure blockchain network using the appropriate technologies [2-4].

The automatic execution of smart contracts without human interaction is another aspect. It will be feasible to execute it if specific requirements are given for its operation in the blockchain network. Any programming error the developer makes can impact the application program, user data, financial resources linked to the program, and even the users' financial resources. A smart contract flaw can occasionally impact the blockchain network infrastructure [2-5]. Smart contracts feature a number of security flaws, some unique to this kind of network and some derived from broader ideas and applied to this particular kind of technology [6-9].

One of the first and most active blockchain networks is Ethereum, which powers smart contracts using a unique programming language called Solidity. It is why this study's focus is on Ethereum and the aforementioned language [10].

After reviewing related literature for this study, we concluded that the following questions still require a response. The current research has attempted to answer these questions in this regard.

- Which particular security flaws exist in smart contracts?
- What is a thorough description of these circumstances that includes examples?
- Is it possible to classify vulnerabilities?
- What methods are available to address these vulnerabilities?
- Which vulnerabilities are verified through the use of various techniques for static code analysis and a comparison of the outcomes in smart contracts?
- Do these vulnerabilities occur more frequently in some smart contracts than others? Or are some more prevalent?

For this investigation, a huge variety of sources were initially examined. The results were made to attain sufficient value and comprehensiveness by analyzing a pruning operation about the Ethereum (ETH) blockchain network, the unique smart contract vulnerabilities, and the Solidity programming language. The second stage involved looking for flaws in 100 smart contracts active on the publicly accessible Ethereum network until the study time, the number of major transactions, and the financial resources associated with four static code review tools. After evaluation and agreement, the outcomes were presented as a statistical analysis of the outputs. An extensive investigation of the specific Ethereum smart contract vulnerabilities has been attempted in this study. The list below shows the main innovations of this research.

- A thorough examination of particular Ethereum smart contract security flaws with pertinent examples and solutions
- Using several automatic tools to perform static analysis on a collection of smart contracts and provide statistical analysis of vulnerabilities found.
- Establishing a category for smart contract weaknesses.
- A description of fundamental blockchain ideas and comparison data

The smart contract security problem inspired us to write this paper. After creating smart contracts, many developers become aware of security issues, but because of their unchangeable structure, they must construct a new contract in order to address them. This is quite expensive. We have attempted to review the most significant issues and current solutions in this study by looking at the security issues with smart contracts and statically analyzing the codes of a sample of them. To determine the status of these vulnerabilities in actual contracts, we additionally analyze a set of contracts' weaknesses in terms of static code review. The rest of this paper organized as follows:

Section 2 describes related works. Section 3 introduces known vulnerabilities and their detection process, and also covers countermeasures. Based on the flaws described in Section 3, a statistical analysis of a dataset of Ethereum smart contracts has been presented in Section 4. Conclusions and recommendations are included in Section 5.

2. Related works

Related work falls into one of three categories:

- **Category 1: Research that only addresses one or a few specific vulnerabilities does not provide a comprehensive view of the security vulnerabilities of smart contracts.**

In [11] Wöhrer and his colleagues examined six security patterns in Ethereum smart contracts. Of course, it should be highlighted that these instances use the same smart contract security flaws that are described in this study in a different form. In [12], the researchers investigated re-entrancy vulnerability in a data set of smart contracts. This review consists of two review sections with several automated tools and manual reviews. According to the research results, the difference in these cases is very high, and this shows that automatic tools have many false positive reports. In [13], researchers have introduced various possible vulnerabilities in smart contracts. In this research, a wide range of more general vulnerabilities from the level of smart contracts have been proposed. Failure to provide examples and solutions for vulnerabilities is one of the problems of this research. Researchers in this study have also investigated some vulnerability detection methods.

- **Category 2: Research that covers a decent number of existing vulnerabilities but does not include providing solutions or examples in sufficient detail.**

Researchers have presented nine security vulnerabilities in reference [14] without offering a fix or outlining the detection mechanisms in these situations. Then, False/True positive and False/True negative criteria were developed to evaluate the abovementioned vulnerabilities. Without discussing the fix, researchers in research [15] published a list of Ethereum smart contract vulnerabilities (some briefly and others in detail). After assessing a vulnerability and offering a solution, they covered several automatic checking technologies. In [16], researchers created a dataset of Ethereum smart contracts and labeled its vulnerabilities with the outputs of five automatic detection tools. There are no descriptions of vulnerabilities or recommendations for solutions in this study. Furthermore, many of these contracts are no longer in use because they have expired. The research's methodology entails scanning a set of smart contracts that were amassed by another study [17] and classifying the data according to the results. In [18], the research method served as the study of publicly available scientific sources and their elimination to achieve the research objective. With a sample contract and very brief explanations. Researchers outline three areas of smart contract weaknesses in [19]: blockchain, code, and Ethereum virtual machine. In this approach, some vulnerabilities are described with examples, while in others, merely mentioning the subject is sufficient. It should be highlighted that no remedy was discovered in this study, and other more general and relevant smart contract vulnerabilities have also been suggested. In the research [20], the researchers have investigated a wide range of vulnerabilities related to smart contracts in the form of a general classification in terms of the impact and severity of the vulnerability. Of course, many of these cases are not specific to smart contracts, and a wide range of software also faces them. Limited explanations in describing vulnerabilities, examples and related solutions are another feature of this research. Researchers have also examined developers' views and provided some general solutions for diagnosing security problems.

- **Category 3: Research that has concentrated more on static code analysis has not taken any action to improve the accuracy of the static analysis output or statistically assess the findings, in addition to not giving enough details about each vulnerability.**

The study's authors [21] have also looked at thirteen other smart contract weaknesses and given corresponding solutions, limited to the Ethereum blockchain and following the current weaknesses through the review of English-language scholarly sources. An overview of vulnerability detection tools and their comparison was then completed. In the study [22], the researchers presented, reviewed, and contrasted the techniques for identifying these vulnerabilities in static and dynamic methods (the research's primary focus) after providing a very brief description of 11 major smart contract flaws. In the research [23], the researchers have investigated a set of Ethereum smart contracts (related to the SmartBugs tool) with nine automatic static vulnerability-checking tools. In this research, only the warnings related to automatic tools have been investigated, and no proper effort has been made to increase the accuracy of the results. For this reason, the number of vulnerable contracts is reported to be very high. In the research [24], after pointing out the vulnerabilities of smart contracts, the researchers introduced the available automatic detection tools and categorized and compared them. In this comparison process, things such as the speed and accuracy of diagnosis, the amount of error in diagnosis, and the types of interfaces that can be used have been examined. According to the research methods utilized in three studies [16,24,25], valid scientific papers were found through an academic search, and they were then culled based on the requirements of the study questions.
The second and third category studies may not be able to provide a proper view for the reader due to insufficient information. In the third category, the accuracy of the results can also be very poor because the output of error tools is usually accompanied by many erroneous reports. Therefore, it is necessary to provide appropriate analyses and sufficient information in addition to paying attention to the issue of increasing accuracy.

3. Research background

Some useful and crucial concepts related to smart contract vulnerabilities are provided in this section.body.

3.1. Blockchain network

Blockchain is a decentralized digital collective agreement that uses one-way encryption technology or hashing to stop information from being changed or manipulated. Due to the limited public access to this disseminated information, this network also exhibits transparency. These networks maintain duplicate copies of transaction information—information recorded in the network and made up of numerous blocks—between dispersed nodes. The likelihood of data loss will be significantly decreased in this approach. Some blockchain terminology is defined in the sentences that follow [8], [24-25].

3.1.1. Blocks

The blockchain network collects transactions in the form of blocks over a predetermined period. The following data [24-25] is saved in a block next to the transactions mentioned above **:**
Magic number: used to identify a block as a certain digital currency's network component**.**

- **Block header**: provides details about the block.

- **Block size**: specifies the maximum block size for writing a given data **.**
- **Transaction Counter**: A transaction counter is a number that indicates how many transactions have been stored in a block**.**
- **Transactions**: Lists each transaction that took place within a block.
- **Version**: Displays the current version of the digital currency.
- **Preceding block hash**: Contains the preceding block's header's hash (an encrypted number)**.**
- **Root Merkle Hash**: The current block comprises the hash codes of the transactions in the Merkel tree, a tree structure designed to minimize storage capacity and transaction calculations.
- **Time**: includes the time stamp of the blockchain block submission**.**
- **Bit**: Contains information describing the complexity of finding the nonce number and the difficulty of the target hash function **.**
- **Nonce**: A random number that the miner must find to validate and close the block.

An abstract block structure from the blockchain network is shown in Table 1**.**

Table 1. Block structure [21]

Block Version	0200000
Parent Block Hash	b6ff0b1b1680a2867a30ca44d34d9e8910d334be b48ca0c000000000000000
Merkle tree Root	9d10aa52ee949386ca9385695f04ede270dda208 10decd12bc9b048aaab31471
Time Stamp	24d95a54
nBits	30c31b18
Nounce	Fe9f0864
Transaction Counter	
TX1 TX2 TX3 ……………………….TXn	

3.1.2. Validator

According to the decentralized collective consensus protocol, the systems in the blockchain network are in charge of validating and logging network transactions. The network validators are rewarded for completing these duties. It should be highlighted that any actions taken by validators that violate the blockchain's rules are detected and removed by the decentralized collective consensus protocol [25-27].

3.1.3. Consensus protocols

In the blockchain, a decentralized consensus protocol allows different nodes to agree on a network element (blocks). The blockchain network's nodes all concur to record the same data according to mutual understanding. Without the assistance of a centralized organization, this activity is carried out decentralized. We will look at the most well-known decentralized collective consensus protocols introduced [25-26].

- The POW (Proof of Work) protocol pits high-powered systems against one another in a race to crack tricky computational riddles. The first miner to find the solution, or, to put it another way, a hash compatible with the prior blocks, sends it to the network for confirmation by other nodes. It is better to describe the hashing method to comprehend this problem briefly. There is a form of one-way encryption used in the hashing method. This means that if X is present, a specific hashing technique can be used to quickly determine the value of h(x), although this problem requires a lot of computation and is expensive. The POW protocol reward miners to use trial and error to develop h(x) solutions compatible with the earlier blocks. This allows the miner who solves h(x) to broadcast it to the network first, where other miners can quickly verify it and validate its accuracy [26-28].
- The POS protocol (proof of stake) eliminates the prior form's flaw of requiring high processing costs and electrical resources from the verifiers. One of the network nodes is selected randomly (in a transparent procedure), with no longer any competition amongst the block verifiers, to validate the block. People with a higher amount in these two parameters have more chances in this selection process, which is based on the number of shares staked and the length of time the staked capital is locked. Age, a protocol parameter that describes the nodes' waiting time, is present in this protocol. This value will be zero if a validator is chosen to confirm the block,

increasing the likelihood that a fair procedure will involve waiting for validators. A validator receives a bonus in the form of the network's native currency after validating a block and submitting it to the network [26-27], [29].

- Proof-of-authority protocol: This protocol, which aims to utilize fewer computational resources than the POS protocol, is an alternative. This approach bases agreement on Proof-of-authority. This means that validators are chosen using a strict method, and then the accuracy of those validators is evaluated using a monitoring standard. A validator loses his validity and is removed from the system if his performance falls short of the required level. As a result, the right to validation is directly impacted by standard compliance. The identity of the validator, his history, and the amount of investment are all examined during the selection process [30-33].
- The enhanced POS Consensus Protocol is DPoS (Delegated Proof of Stake). By putting their tokens (the principal infrastructure blockchain token) into an investment pool, users in this system can cast votes for validators which will approve and create the block. Naturally, more deposits increase power in a way that prevents the system from being dominated by one or a few actors. Depending on the proportion of tokens invested, the validator will distribute a portion of the profit to the voters if the block is written correctly. By garnering more votes, more trustworthy and accepted validators are given more authority to change the block. It should be highlighted that only a few validators can vote and are chosen to write blocks. Naturally, this list is flexible [34-37].
- PBFT protocol (Practical Byzantine Fault Tolerance) Even with faults in specific dispersed nodes, this system may reach a consensus. Assume that the PBFT protocol allows a distributed system with 3f+1 nodes to have a maximum of error nodes (Byzantine nodes). In other words, when 2f+1 nodes agree on a message, the system as a whole will likewise agree. In this system, the process of republishing and confirming these messages in the form of a "preparation message" in the network is carried out by a set of pre-defined messages. These messages are called the first preparation message that is transmitted from a node called Primary Node to several additional nodes called Replicator. Validator nodes can advertise their ability to write the current block by broadcasting the "confirmation" message after receiving 2f+1 messages compatible with their sent item. It is done by rebroadcasting preparation messages. When the first node receives 2f+1 compatible messages of this type, It sends the confirmation message in the following step, known as the commitment step, writes the block, and modifies the system state [38-40].

The comparison of the blockchain consensus protocols is shown in Table 2 [41-43]. Table 2 indicates that every methodology has pros and cons of its own. Some of them have been improved from this perspective and outperformed POW in terms of speed, energy consumption, and scalability. Of course, in these cases, security weaknesses, power distribution, or increased communication overhead are seen.

3.1.4. Blockchain generations

In general, three different generations are defined in the blockchain. In the first generation (the first form of blockchain), network miners expended much energy and money to confirm decentralized transactions without a central processing unit. A prolonged process that was completed peer-to-peer and without the use of middlemen. The Bitcoin cryptocurrency project, of course, had fewer modifications from this generation than the first, but second-generation blockchains have undergone major alterations since the first generation. Deploying apps with smart contracts and asset management on the blockchain became possible with the second-generation blockchains. The second generation's key features also include the ability to issue blockchain shares (along with money and smart wallets). In comparison to the prior generation, transaction speeds also significantly improved in this generation [44-48].

Low scalability and security issues were caused by high prices and slow speed. For this reason, new blockchains emerged to solve this problem under the title of the third generation. The critical features of this generation of blockchains are simplicity in data access, better scalability, and higher speed. The third generation of blockchains made it possible for blockchains to connect with one another, which was also impossible in the prior two generations. The layer definition is another characteristic of the third generation of blockchain technology, which has considerably increased both the speed and security of the system [44-46].

3.2. Smart contracts

On the blockchain network, smart contracts are deployed and carried out as automated applications that are programmed. If the required conditions (described in the code) are established in the blockchain network, this contract can operate autonomously. Naturally, smart contracts must pay a GAS fee or network usage fee to register information in the network to prevent excessive consumption of blockchain network resources. The information in the blockchain network is disseminated among all users, as was indicated in the previous section. It is no longer possible to update information once it has been published. In other words, once a smart contract has been executed and has reached its expiration, it cannot be

stopped. Gas fees are particularly important in reducing resource usage. To put it another way, they are employed to stop malicious users from abusing the blockchain network's limited resources by creating fictitious transactions to take up available memory, processing power, and other resources. Additionally, by using this strategy, network management can get financial resources that they can utilize to grow, enhance performance, or better administer the network. [1-2], [5], [49-50].

After programming in the blockchain and using the VM (or executive virtual machine), the codes for a smart contract are performed. Then, to interact with it, users must carry out a specific transaction. With the aid of this transaction, users sign the smart contract and validate its usage within the blockchain network using their private key and asymmetric encryption. After the smart contract has been signed and verified, users can interact with it and use it (for instance, purchasing the tokens provided by the smart contract using tokens associated with the native blockchain network). Usually, specialist programming languages like Solidity or Serpent are used to create smart contracts. Of course, common languages such as Python are also active in this field. These kinds of languages offer particular functionality utilized by blockchain in addition to the standard characteristics provided by programming languages (concepts like variables, functions, classes, computing operations, and strings) [1-2], [5]. Some of the most effective uses for smart contracts in use today are depicted in Figure 1.

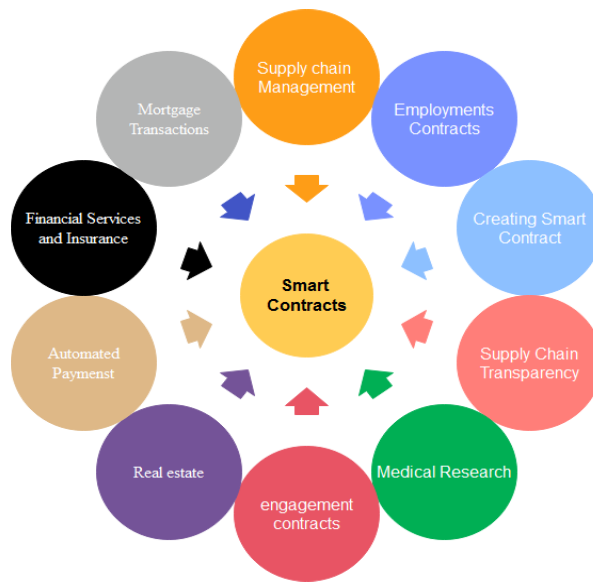

Figure 1. Some of the most important applications of smart contracts today [51][†]

[†] All figures generated by Edraw Max Tool

Table 2. Comparison of consensus protocols

Consensus Protocol	Advantages	Disadvantages
POW	• High network security • Creating more employment	• High energy consumption • The need for high-processing equipment • High cost • The possibility of network control falling into the hands of a specific group, especially in smaller networks • Ability to control the network by a specific group • Moderate scalability
POS	• lower cost • Less energy consumption • Higher speed	• The possibility of greater vulnerability • Increase concentration • Ability to control the network by a specific group • Moderate scalability
DPOS	• Cost optimization and energy consumption • Increased processing speed	• Vulnerability to some network attacks due to increased communication
POA	• Lower transfer cost • Good security • High scalability	• Hard implementation • Good protection against denial of service attacks • Less distribution
PBFT	• Low energy consumption • High processing speed • Low cost	• High communication overhead • Vulnerability to some network attacks due to increased communication • vulnerable to denial of service attacks • High communication complexity

Table 3. shows the characteristics of different blockchain generations comparatively

Blockchain	Advantages	Sample projects	Disadvantages
Generation 1	Distributed transactions without intermediaries	Bitcoin	High cost, power consumption, low speed, poor scalability, and security issues
Generation 2	Better speed than 1st generation, distributed, transactionless transaction, smart contracts, smart money, and wallet.	Ethereum, Ethereum Classic, and Neo	Low scalability, security issues, lack of proper communication between blockchains
Generation 3	Distributedness, transaction without intermediaries, smart contracts, money and smart wallet, inter-blockchain communication, simplicity in data access, layered structure, better .security, and higher speed	Cardano, Algorand	Security issues, privacy issues, and .lack of global policies

3.3. Solidity programing language

One of the most well-known programming languages for smart contracts is called Solidity. This high-level, object-oriented language is compatible with Ethereum, Solana, and Binance Smart Chain, among other blockchains. In the Solidity programming language, a smart contract is made up of both data and code. Copies of the smart contract are placed on different network nodes, and it is given a public address (connected to public key cryptography). After that, users register their transaction for network execution by calling the contract using this address. Solidity smart contracts are carried out on the Ethereum network's virtual machine. With vast memory and processing resources, this virtual machine is a full Turing computer for mathematical calculations [16], [52-53].

3-4. Security Vulnerabilities

A vulnerability in smart contracts refers to a kind of security weakness that can satisfy the three rules in the list below. The existence of vulnerability will also be violated if any one of these three situations does not present [54]

- a flaw in the system that puts information security at risk.
- The potential for an attacker to access the pertinent defect
- The potential for an attacker to exploit that defect.

4. Vulnerabilities in smart contracts

In this section, we present the vulnerabilities in smart contracts along with the appropriate countermeasures. This section mostly addresses smart contract vulnerabilities. Things that are universal from a broader idea have not been included.

4.1. Re-entrancy vulnerability Check

4.1.1. Introduction of reentrance vulnerability

Reentrance vulnerability describes an external contract's ability under a cyber attacker's command to call contract functions. It may result in the depletion or intrusion of Ethereum (smart contract inventory). In other words, this vulnerability occurs if a smart contract uses the call, share, and send functions to transfer flow control to another smart contract, then updates the status of the primary contract using the callback function (in this case, the smart contract status may be "Not completed"). The smart contract is incomplete under the circumstances above (the callback function does not return the expected value), and the calling contract may run further smart contracts or functions based on the written codes. Pay attention to Sender's smart contract code [7] ,[55-57] for a better understanding.

Code snippet 1: Reentrancy Vulnerability

```
pragma solidity >=0.4.22 <0.6.0;
contract Sender {
uint public amount;
address payable public sender;
address payable public receiver;
 constructor() public payable {
 sender = msg.sender;
 amount = msg.value;
}
function send(receiver) payable {
receiver.call.value(value).gas(20317)();
}
}
contract Receiver {
uint public balance = 0;
function () payable {
balance += msg.value;
}
}
```

Ethereum is transmitted to the recipient address, and the callback function is activated when this contract's send function is used. To implement this function, the average consumption of GAS is 2300 [58], which is received even in the incorrect implementation. While the amount of gas in the transfer and call functions is altered or left unrestricted, they are similar to this function in other ways. It should be noted that GAS is not deducted in the transfer function when a problem occurs. code snippet 2 has a related vulnerability [55].

Code snippet 2: Reentrancy Vulnerability 2

```
function transferBalance(address receiver, uint
amount)
public {
require(balances[msg.sender] >= amount);
receiver.transfer(amount);
/* flow control transferred before the sender'
s balance is updated before an event
is emitted. Potentially the start of
trouble. */
balances[receiver] -= amount;
LogTransactions(msg.sender,receiver, amount);
}
```

As can be seen, this contract's balance is transferred through the transfer function. After executing this function, the user's inventory status will be updated (in actuality, the contract status will change). By calling this method and modifying the callback function in this scenario, the malicious smart contract can empty the Ethereum balance of the smart contract [55]. When two functions or smart contracts share a state, a reentrancy attack can still succeed. For instance, the code below demonstrates this. Because of the value of balance[msg.

Sender] is not set to zero in this contract after the reentrancy attack. The transfer function in the code snippet 3 can use this variable to send additional Ethereum by calling the call function in the withdraw function [55]. The general layout of the reentrancy assault is depicted in Figure 2.

Code snippet 3: Reentrancy Vulnerability 3

```
mapping (address => uint) private balance;
function transfer(address to, uint amount) {
if (balance[msg.sender] >= amount){
balance[to] += amount;
balance[msg.sender] -= amount;
}
}
function withdraw() public {
uint amount = balance[msg.sender];
require(msg.sender.call.value(amount)());

/* At this point, the caller's code is
executed and can call transfer() */
balance[msg.sender] = 0;
```

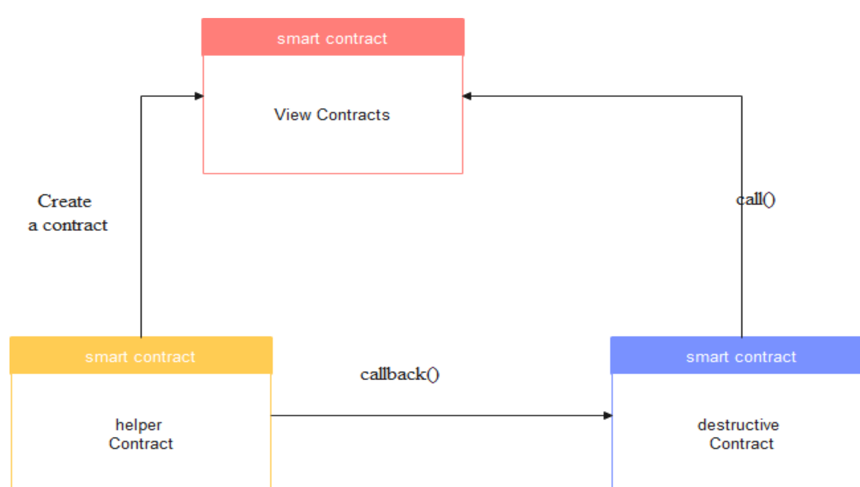

Figure 2. The overall structure of the Reentrancy attack [59]

4.1.2. The Solution to Reentrancy vulnerability

Changing the status or value of the influencing variables should be transferred before invoking the call, transmit, or send functions, as was stated, to address this issue. The reentrancy attack can no longer be successfully undertaken in this situation [55].

4.2. Investigating the vulnerability of access control in smart contracts

4.2.1. Access control vulnerability

User roles and privileges within an application are restricted using access controls. This notion in smart contracts can be connected to governance and crucial logic, including token issuance, proposal voting, money withdrawals, suspension and contract upgrades, ownership changes, etc. The access control mechanisms used by intelligent contracts have some flaws. The following can be listed as weaknesses among them [7], [9], [60].

Non-Validation of Modifiers: In critical functions, validating modifiers is crucial. The owner can be changed, tokens and existence can be transferred, and contracts can be stopped and canceled, among other things, thanks to modifier functions. Modifiers' access levels to key functions must be verified before use to avoid risks like money loss or contract termination.

Incorrect Modifier names: A Modifier's name or even the name of a function may be written differently than what is specified in the validation library due to programmer error. As a result, neither function nor modifier is subject to the modifier process anymore. Which, depending on performance, may result in a loss of cash or a change of ownership.

Having too many roles defined: Enabling users to have an excessive number of roles could result in an access control vulnerability.

Take a look at the example below for a better understanding.An access control problem that allowed anyone to burn Hosp tokens led to the hacking of the Hospowise project a while back. This smart contract's

susceptible token burning function is related to the burn function [61].

function _burn (account, amount); public; burn(address account, uint256 amount);

Put another way. A hacker might buy any token and then use the public write function. This function burns all of the Hospo tokens on the UniSwap exchange. The attacker can now trade his tokens for the Ethereum cryptocurrency because the token's value has increased [57].

4.2.2. Access control vulnerability solution

The process needing access control permissions must be reviewed to address this kind of vulnerability. This can be used by smart contract programmers with specialized access management libraries. The name of one of these libraries is OpenZeppelin. This library's Openzeppline's Ownable subsection offers modifiers like "only owner" to ensure that a function is called by its owner. It should be noted that other access control management sub-sections in OpenZeppelin employ modifiers or functions like "hasRole" to determine whether a user has the authority to invoke a function [62]. There are also a few appropriate models for controlling access in smart contracts [63].

4.3. Investigating arithmetic vulnerabilities

4.3.1. Introduction of arithmetic vulnerabilities

Smart contracts are extremely vulnerable to integer overflows. Unsigned integers are typically used in smart contracts, and most developers work with simple integer types (often only signed integers). Many code paths can become carriers for information theft or service denial when there is a buffer overflow [9-10], [64,65].
The below smart contract's ability to remove balances serves as a vivid illustration of this vulnerability. This function should permit withdrawals if the balance is greater than zero (and more significant than the minimum transaction cost). It is true even though it is possible to withdraw larger sums of money via the integer overflow vulnerability. In other words, the withdraw () function's check result is always positive, allowing the attacker to withdraw more than the permitted amount. Consider the following function [65] to gain a better understanding of the problem.

Code snippet 4: Arithmetic vulnerabilities

```
function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
}
```

This method's sole criteria to determine whether the function caller has permission to withdraw funds balance is [msg. Sender] - _amount > 0. a procedure that enables free withdrawal in case of a buffer sequence vulnerability and can be bypassed.
The code snippet 5 has an arithmetic flaw and is connected to the Beauty Chain blockchain. A susceptible function named batch transfer is employed in the

Code snippet 5: Arithmetic vulnerabilities 2

BeautyChain smart contract, and it is in this function that the vulnerability mentioned above exists [66].

```
function batchTransfer(address[] _receivers, uint256
_value) public whenNotPaused returns (bool) {
uint cnt = _receivers.length;
uint256 amount = uint256 (cnt) * _value;
require(cnt > 0 && cnt <= 20);
require (_value > 0 && balances[msg.sender] >=
amount);
balances [msg.seneder] =
balances[msg.sender].sub(amount);
for (uint i=0; i<cnt; i++) {
balances[_receivers[i]] =
balances[_receivers[i]].add(_value);
Transfer(msg.sender, _receivers[i], _value)'
}
return true;
 }
}
```

The local variable value code determines the sum by multiplying the cnt and _value variables. A 256-bit integer may be used as the second parameter in this multiplication, designated as _value. Due to the batchTransfer() function's ability to accept two inputs, a hacker might overflow the amount variable and set it to zero by passing an extremely large number in the _value parameter. All checks on colored lines will succeed by setting this variable to zero, and the subsequent line's subtraction won't matter. Finally, the code will dump numerous recipients' balances with a very big value without incurring any fees to the attacker's account [66].
4-3-2. Solution of arithmetic vulnerabilities
Rewriting the susceptible sections of the code using the supplied libraries is a security fix for this vulnerability. SafeMath is one of these libraries, and it can be used to address arithmetic flaws. The safe functions add sub, mu, and div were used to construct the library's four core activities [67]. After including the SafeMath library in your smart contract, swap out all four major operations with the functions mentioned above to address the vulnerability. These alterations are listed below.

Code snippet 6: Arithmetic vulnerability Solution

```
a * b becomes a.mul(b)
a / b becomes a.div(b)
a - b becomes a.sub(b)
```

For example, the add function in this library is as follows.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
    return a + b;
  }
```

You can see that the result is returned as a pure function and uint256. It is guaranteed not to read or modify the status information in the pure function.

4.4. Not checking return values in low-level calls

4.4.1. Challenge return values in low-level calls

The availability of low-level Solidity functions like call(), call code (), delegate call (), and transmit() is one of the aspects of programming languages for smart contracts. In smart contract coding, it is often preferable to employ alternatives to low-level calls whenever available. Incorrect calling of these functions could result in the contract's security being jeopardized. These functions in Solidity behave entirely differently from the rest when it comes to accounting for errors. In other words, the error in a function of this kind is not even broadcast and does not result in a full return from the present execution. In other words, these functions only ever yield the logical value false, after which the Ethereum virtual engine is still executing the code. Consider the example in [9], [68] to better understand the problem.
Using the recipient's wallet address is the most straightforward approach to creating a contract to send Ethereum to another address in this blockchain network. The following conditional check (Code snippet 7) is a game component on a board where the winner receives money in the form of Ethereum digital currency [68].

Code snippet 7: Not checking return values in low-level calls

```
if (gameHasEnded && !( prizePaidOut ) ) {
  winner.send(1000); // send a prize to the winner
  prizePaidOut = True;
}
```

The issue is that the sending procedure—for instance, the send function in the Code snippet above—might not succeed. The prizePaidOut variable will be set to True, and the winner won't get any money. This value shouldn't be set to true because the correct transfer has not yet been made. The leading cause of this problem is that the programmer neglected to examine the send function's output. There are two situations in which the winner.send() function might not work as intended. First, an exception is made when the winning address relates to a contract rather than a user account. In this instance, the failure to mail the award is the winner's responsibility because they provided an erroneous address. In the second scenario, other contract programs (which have already been executed in the transaction) can use a finite resource called "Callstack" that is available on the Ethereum virtual machine. This operation will fail (and the winner's award will be forfeited) if this resource is used up before transmitting Ethereum to the winner's address. The smart contract must be appropriately secured to protect the winner. It should be mentioned that depending on the type of smart contract code, the results of this vulnerability may be extremely dangerous. For instance, by altering the values of the variables without providing any inventory, a cyberattacker could purposefully improve his access level to the smart contract [68-69].

4.4.2. Resolving return values in low-level calls

Two ideas are mentioned below [68] as a means of addressing this vulnerability:
Examining the results of low-level functions: This method uses vulnerable functions to check the output linked to vulnerable spots before using it if there are no issues. The following code snippet fragment provides a viable solution for the previously mentioned example using the same methodology.

Code snippet 8: Not checking return values in low-level call solution

```
if (gameHasEnded && !( prizePaidOut ) ) {
  if (callStackIsEmpty()) {
  if (winner.send(1000)){
    prizePaidOut = True;
}
  else throw;
} else throw;
}
```

In this example, the stack's emptiness is verified using the callStackIsEmpty method, and only if there are no issues is the sending procedure started and the prizePaidOut variable set. You should modify your code to ensure that the outcomes of unsuccessful sending are distinguished from other scenarios [68].
Utilizing exceptions Making use of exceptions is another method to address this problem. By specifying an exception in this method, the programmer can take control of unusual circumstances, including the results of not inspecting the return values in low-level calls. In other words, when the circumstances mentioned above occur, the program's execution flow departs from normal mode, and the code relating to the exception is run.

4.5. Investigation of denial of service attacks

4.5.1. Denial of service attacks

A denial of service attack is one in which the attacker sends the victim's system numerous input requests to deny service to the main users. Smart contracts may be permanently shut down following a denial-of-service

assault, in contrast to web services that can occasionally recover from these attacks. Denial of service attacks can be implemented in smart contracts in a variety of ways. The following list of instances can be listed among these [7], [70-71].

- Malicious behavior when receiving a transaction
- Artificial increase of Gas Fee necessary to calculate a function
- Abuse of access controls to access confidential parts of the smart contract

Take a look at the example below for a better understanding. The code snippet 9 relates to the blockchain game King of the Ether, which lets players buy access to the president through payments made in Ethereum to other players. The access transfer process will now fail if the other user is a smart contract, and the stated smart contract will always have the president's access authorization. Based on this, a denial of service attack has been conducted due to faulty access authorization management [70].

Code snippet 9: DOS attack

```
function becomePresident() payable {
   require(msg.value >= price); // must pay the price to
become president
   president.transfer(price);   // we pay the previous
president
   president = msg.sender;      // we crown the new
president
   price = price * 2;           // we double the price to
become president
}
```

Additionally, the smart contract may be terminated if the GAS Fee for a transaction exceeds the maximum amount allowed in a block because other network transactions will not be able to be accepted. A smart contract may unintentionally cause it to occur, as well as online hackers. Consider a cyber attacker who creates many destination addresses to steal modest amounts from the victim's smart contract. The following function is where this kind of assault can be carried out. The code even states that many current transactions may cause the smart contract to break at some time, stopping all future transactions. The function iteration in the loop is based on the current length of the address array. It can allow an attacker to exhaust resources by adding new return addresses to the array [70]. Additionally, since the refund function's transaction depends on the fixed sending function, which has a value of 2300 GAS, it is possible to artificially increase the GAS fee in this function.

Code snippet 10: DOS attack 2

```
address[] private refundAddresses;
mapping (address => uint) public refunds;
function refundAll() external onlyOwner {
// unknown length iteration based on how many addresses
participated
for(uint i; i < refundAddresses.length; i++) {
// doubly bad, now a single failure on send will hold up all
funds
require(refundAddresses[i].send(refunds[refundAddresses
[i]]))
}
}
```

4.5.2. A solution to Denial of Service attacks

The smart contract's codes should be securely examined, particularly in the control discussion, to defend against denial of service attacks (checking the actions of all departments and access permissions). Before deployment in the blockchain network, these codes must be rewritten if there are any issues. Additionally, the ability to interact with the smart contract should be limited, especially during procedures like the withdrawal of funds, and appropriate thresholds should be set to regulate the volume of requests. This procedure should be carried out within the block limit for the GAS Fee. Instead of using the transmit() method in certain circumstances, utilizing a transferFrom() function is preferable.

4.6. Failure to use appropriate random numbers

4.6.1. The challenge of generating random and pseudo-random numbers

In general, using actual random numbers is exceedingly challenging and impossible. Cyber attackers can also duplicate numbers and take advantage of the system, regardless of how far this process of creating random numbers is from reality. Despite the widespread use of random numbers (used in many programs such as lotteries, games, airdrops, etc.) and the direct link to financial issues, this issue is of utmost importance in smart contracts [7], [9]. Pay close attention to the lottery game example below to better understand the topic—a lottery based on random numbers where the winner receives cryptocurrency for their prize.

The DiceGame smart contract (Code snippet 11) serves as a prime illustration of this. The user must estimate the smart contract's random number in this code; if successful, he will be rewarded with one Ethereum. It is carried out using the random() function and the guess_the_dice() function. The random() function creates a random number by using the previous block's block number and the current block's timestamp [72].

Code snippet 11: Random Number Problem

```
contract DiceGame
{
constructor() payable{ }
```

```
function guess_the_dice(uint8 _guessDice) public {
uint8 dice = random();
if (dice == _guessDice) {
(bool sent, ) = msg.sender.call{value: 1 ether}("");
require(sent , "failed to transfer");
}
}
// source of randomness (1-6)
function random() private view returns (uint8) {
uint256 blockValue = uint256(blockhash(block.number-1
+ block.timestamp));
return uint8(blockValue % 5) + 1;
}
}
```

Looking at the smart contract's random function, it is obvious that a hacker could create this number using the following function and drain the balance of the account by guessing the winning lottery number [68].

Code snippet 12: Random Number Problem 2

```
function random() private view returns (uint8) {
uint256 blockValue = uint256(blockhash(block.number-1
+ block.timestamp));
return uint8(blockValue % 5) + 1;
}
```

4.6.2. The solution to the challenge of generating random and pseudo-random numbers

You should not use publicly accessible and predictable information, such as details about blocks (also known as Onchain) in the network, to defend against this attack. Additionally, it is preferable to generate random numbers using the provided libraries and special functions. One of the most crucial tools for this problem is libraries that offer VRF (Verifiable Random Function), offered by businesses like Chainlink. Smart contracts can produce statistically random numbers using these libraries. With one-way and asymmetric encryption, this method may verify the legitimacy of the random number's creator. It will cause anyone can identify the random numbers produced outside of the smart contract [73-74].

4.7. Forward transaction attacks

4.7.1. The challenge of forward transaction attacks

Transactions in the blockchain network are typically not immediately recorded. In fact, following the registration request, a collection of these transactions must be compiled into a block and come to a consensus in the network. Before choosing a block, all nodes in a network must be informed of transaction details due to the distributed nature of block creation. In other words, when a node in the blockchain creates a transaction, the pertinent information is broadcast to other nodes. After obtaining the indicated info by nodes, they place the transactional information in the "an unused pool" structure. The block creator node adds transactions to the block with the priority of the paid fee amount (the block creator seeks more profit) after adding a sufficient number of transactions (the size of writing a block) to this pool [64], [75-77]. Blockchain network users can prioritize their transactions by paying a more significant price. A cyber attacker can exploit this problem and elevates his transaction above other cases by charging a hefty fee for it. In smart contracts where timing is crucial (like NFT purchase competitions, first response competitions, etc.), it is possible to get around other users in this fashion [58-59]. The Findkey smart contract deserves your attention (Code snippet 13) [78].

Code snippet 13: Forward Transaction Attack

```
contract Findkey {
  bytes32 public constant key =

0x564ccaf7594d66b1eaaea24fe01f0585bf52ee70852af4ea
c0cc4b04711cd0e2;
  constructor() payable {}

  function guess(string memory solution) public {
    require(key ==
keccak256(abi.encodePacked(solution)), "Incorrect
answer");
    (bool sent, ) = msg.sender.call{value: 5 ether}("");
    require(sent, "Failed");
  }
}
```

Finding the value of a string will earn users 5 Ethereum in this smart contract. Let's say user number 1 is successful in figuring out this number and submits it in response to the guess function. The hacker in the blockchain network is currently looking at the transaction data in the "unused pool." He can immediately register it in the blockchain network with a greater registration charge after recognizing the amount in the transaction relating to User Number 1. By using the precedence of transactions in this way, it is possible to get around user number 1 and win 5 Ethereum in its place. You can see how to generally carry out the prior transaction assault in the Figure 3.

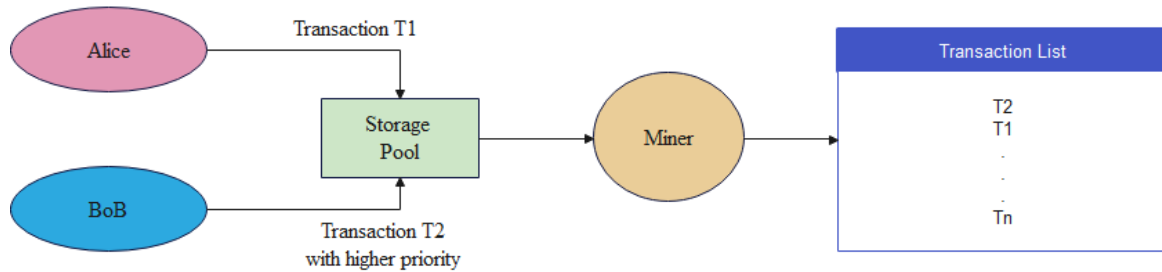

Figure 3. How to execute the front-end transaction attack [79]

4.7.2. Solution of forward transaction attacks

The use of the Commit-Reveal method is a viable answer to this issue. Using encryption techniques, user number 1 in the previous example sends his message locked and immutable over the network. After registering on the blockchain, he can unlock it. Because the attacker cannot see the transaction information before it is registered in the network, the forward transaction attack is no longer viable. The Submarine Send technique is another approach that experts have suggested to handle this predicament. This method involves sending the smart contract's related response to a third address in the blockchain network, encrypted using asymmetric encryption and viewable by the smart contract. The transaction details will then be made available to the smart contract by the sender, who will also disclose this transaction to it. Using this technique prevents the cyber attacker from seeing the transaction data in the collection of unused transactions [80-81].

4.8. Wrong time dependency vulnerability

4.8.1. The challenge of wrong time dependency vulnerability

The programmer may need to precisely record the time in smart contracts to use it in the program's logic. This value is called the time stamp. Any error in the definition of this topic could lead to a security flaw. This issue might arise, for instance, if the programmer utilizes a seal associated with the most recent block in the infrastructure blockchain network. One of the most difficult problems in this area is distributing an accurate clock among the nodes of a blockchain network. Therefore, the timing gap between nodes in a blockchain network is correct.
Additionally, this value may be modified by a node or smart contract [7], [9], [82-83]. Study the Code snippet for the EtherLotto smart contract to gain a better understanding of the problem [84]. This lottery application uses the smart contract code. Users must prepare a certain sum equal to the "competition ticket" and deposit it into the smart contract account to take part in this competition. Following (code snippet 14) the generation of a random number, if its value equals zero, the corresponding user will win the lottery. The matching award will then be sent to his address when this occurs.

Code snippet 14: Wrong Time Dependency

```
contract EtherLotto{
  uint constant TICKET_AMOUNT = 10;
  uint constant FEE_AMOUNT = 1;
  address public bank;
  uint public pot;
  function EtherLotto{ ()
    bank = msg.sender;
}
  function play() payable}
    assert(msg.value == TICKET_AMOUNT);
    pot += msg.value;
    var random = uint(sha3(block.timestamp)) % 2;
    if (random == 0){
      bank.transfer(FEE_AMOUNT);
      msg.sender.transfer(pot - FEE_AMOUNT);
      pot = 0;
}
}
```

The use of the block timestamp (block.timestamp) to produce a random value in the play function is the source of the issue in this smart contract. This timestamp is simple to change for the node to have the desired result. For instance, altering this number to 0 can result in the hacker winning the lottery.

Table 4. Changing the value of amount and address in the vulnerability of short addresses

4.8.2. Fixed the wrong time dependency vulnerability

Regarding how to address this vulnerability, there are two options. The first step is to adopt alternate techniques and eliminate vulnerable timestamps. Other secure techniques, such as Chainlink VRF, can be used to obtain the random number for the sample in the case above. Additionally, the timestamp can be obtained in another manner (for instance, by using JavaScript functions like the date() and converting it to the Unix timestamp standard or similar libraries) before comparing the two values. According to

the application's logic, the execution should be terminated if the difference exceeds a specific threshold.

4.9. Vulnerability of short addresses

4.9.1. The vulnerability challenge of short addresses

In general, the Ethereum virtual machine will append many zeros to the end of an address if it finds one shorter than the needed length. By altering the call function values and removing zeros from the end of the destination wallet address, a hacker can exploit this vulnerability [7,64,85] and make an unlawful credit withdrawal from the susceptible smart contract account. Consider the function Send(address, amount) to send a specific amount of cryptocurrency (amount) to the account associated with the address variable in the Ethereum blockchain network to better understand the topic. Below is a sample call to this function [64].
Send(0x1234...67890, 10).

The problem is how to deal with and manage the address and unit values in the smart contract before sending. Assume that these items are assigned values of 20 and 32 bytes in memory, respectively. Now, if a cyber attacker, instead of sending 20 bytes of the address, sends 19 bytes after removing the trailing zeros. With this, the API adds zeros to the amount section after coding. The binary equivalent of the address and the value are placed in the memory in a serialized form. A zero in the memory is added to the address part again (the zero corresponding to the beginning of the amount), but the amount part can be changed to multiples of the desired amount. Table 4 shows how this matter is. According to these values and the limitation in the number of zeros on the left side of the amount value in the memory, it is possible to add two of these zeros to the address value and turn the amount value into a multiple of the original request.

4.9.2. Short address vulnerability solution

Simply include a check function in the smart contract to address this problem. The size of the user's input is verified in this function. In the smart contract for the NonPayloadAttackableToken, in code snippet 15 an illustration of this review is provided. The transfer function's expectations for this checking are two 32-byte parameters and a 4-byte method signature [86].

Condition	Amount and address in memory
Normal Condition	0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f000…001010
Execution of the attack	0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f 00…0101000

Code snippet 15: Short Address Attack

```
contract NonPayloadAttackableToken {
  modifier onlyPayloadSize(uint size) {
    assert(msg.data.length == size + 4);
    _;
  }
 function transfer(address _to, uint256 _value)
onlyPayloadSize(2 * 32) {
   // Do stuff
 }
}
```

4.10. Inventory lock vulnerability

4.10.1. Inventory lock vulnerability challenge

This vulnerability exists if it is feasible to deposit a balance of local blockchain network tokens to the smart contract address without being able to withdraw them. In fact, in the mentioned case, the deposit balance is locked to the contract, and the user can no longer withdraw it. The user may have forgotten to declare the withdrawal function, which can lead to this [9], [87-89]. Pay close attention to the code snippet 16 [90] to better understand the problem.

Code snippet16: Inventory lock vulnerability

```
contract Market {
   function deposit() payable {
   }

   function transfer() {
      uint y = msg.value;
   }
}
```

In this contract, there is only a deposit function, and if the token is deposited to the relevant address, withdrawing is no longer possible.

4.10.2. Inventory lock vulnerability solution

The answer to this vulnerability is simple—you must build a function that will withdraw the deposit balance using the program's logic.
The proposed vulnerabilities are organized into categories in Figure 4. The number of these vulnerabilities can undoubtedly rise. Additional categories, such as [91], are also offered. Unlike previous examples, the proposed taxonomy places the vulnerabilities in smaller groups and does not stop with a list of cases.

5. Statistical investigation of vulnerabilities in a dataset of Ethereum smart contracts

In this section, descriptive statistics are employed. Tables and graphs are examples of data visualization tools used in descriptive statistics, which facilitate analysis and understanding.
The findings of the analysis of a dataset of Ethereum smart contracts with public access to the source code are presented in this section concerning the pertinent vulnerabilities [92]. The prerequisites for the randomly chosen contracts were that they had an adequate number of transactions and the application had been active for at least the previous ten days. The average number of transactions, the time of the most recent transaction, and the total number of contracts are all displayed in Figure 5, the contracts in question. This figure indicates a lot of active transactions for the contracts indicated. The contracts, as mentioned earlier, also contain considerable cash resources.

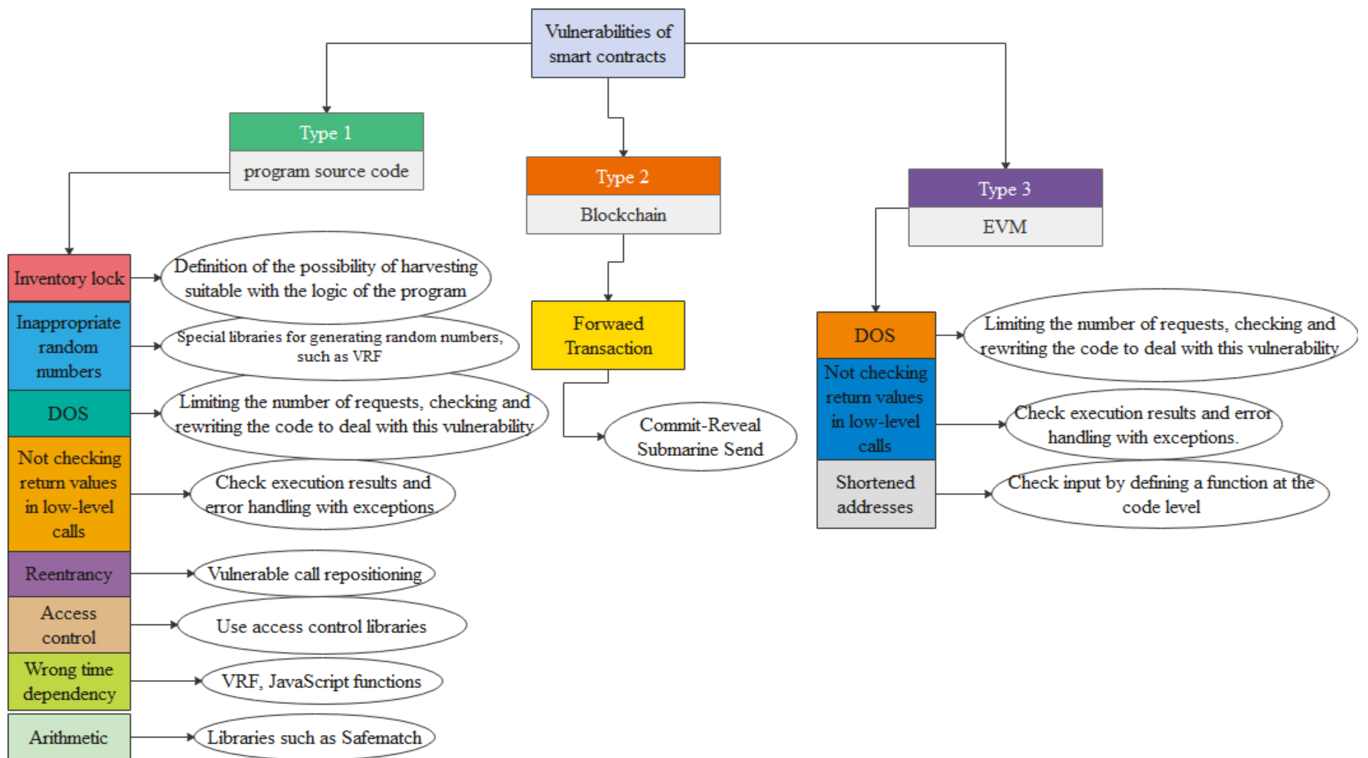

Figure 4. Classification of vulnerabilities related to smart contracts

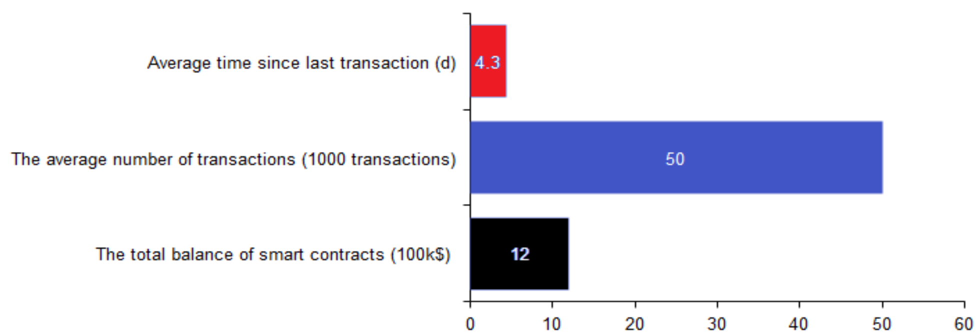

Figure 5. Information of smart contracts reviewed

Our review process includes static scanning at the code level with the help of four automatic vulnerability-checking tools: Slither, Smartcheck, Oyente, and Mythril [92-95]. In this review, a vulnerability warning in other tools is also considered with the help of observing the relevant outputs by each tool. In other words, some verification procedure has also been carried out for vulnerabilities with alerts in many tools. It should be recalled that a vulnerability requires three different circumstances, which were also discussed in sections 3-4. Only when employing the smart contract in the application, or, in other words, the final system, can all these things be investigated. As a result, some of these warnings might be inaccurate (the concept of a False Positive), or certain vulnerabilities might still be present in the system (the concept of a False Negative).

Slither: Slither is a smart contract static inspection tool that can sort the data in the code. This program creates an AST (Tree Syntax Abstract) structure to start, then pulls data from smart contracts like an inheritance graph, flow control graph, and list of Expressions to find the vulnerability. The vulnerability detection procedure is then performed using operations, including checking dependencies, variables, and accesses [93].

Smart check: a dynamic analysis tool for Solidity smart contracts that can be expanded. By transforming smart contract code into an intermediate XML representation and comparing it with patterns gathered from the actual world, this program finds weaknesses [94].

Oyente: A Symbolic review tool specifically made for analyzing smart contract code. This tool uses the CFG Builder (flow control graph maker) and Ethereum Blockchain Explorer modules to deliver the smart contract-related bytecodes in symbolic form to the central review module. This module's vulnerability detection procedure involves checking the smart contract's Symbolic output [95].

Mythril: An Ethereum virtual machine bytecode review tool that employs symbolic analysis, the SMT solver method, and the effect of different inputs on the program with the taint analysis approach to find smart contracts' weaknesses [96].

The reason for choosing these tools is to cover most of the introduced vulnerabilities. In addition, these tools must detect common vulnerabilities to be used in this investigation. Other tools, including Soda and Sfuzz [97-99], have also been published to check the security of smart contracts. However, these cases have been omitted due to the lack of direct coverage of the mentioned vulnerabilities or the lack of sharing in the list of detectable vulnerabilities.

In terms of percentage and diversity of vulnerability alerts, Figure 6 presents comparisons between smart contracts. As shown, at least one vulnerability warning exists at the code level for around a third of the extant smart contracts at the dataset level. This problem demonstrates both the significance of the issue and the lack of sufficient attention given by smart contract developers to the field's vulnerabilities. In Figure 7, you can also see the results of the vulnerability warning check in the data set by the vulnerability. According to the graph of arithmetic-type vulnerabilities, forward transactions and locked ether have the most repetitions among the examined data sets. Based on this, it can be said that inattention to the security of at least one calculation, as well as not considering (or not knowing) the definition of priorities in the infrastructure blockchain network or not defining a method for capital withdrawal from the smart contract are the most common vulnerabilities in smart contracts (at least in this data set). In this way, smart contract developers may employ an equal methodology.

This issue was validated by looking at the coding associated with the vulnerable smart contracts for the three vulnerabilities above. For instance, smart contracts that were flagged for having an arithmetic vulnerability employed unsecured routine methods to perform their computations. In addition, the outputs linked to the tools mentioned above still have many inaccuracies, as we discovered when manually reviewing the vulnerabilities. Put otherwise, and many pertinent reports were false positives since they could not be verified through manual examination.

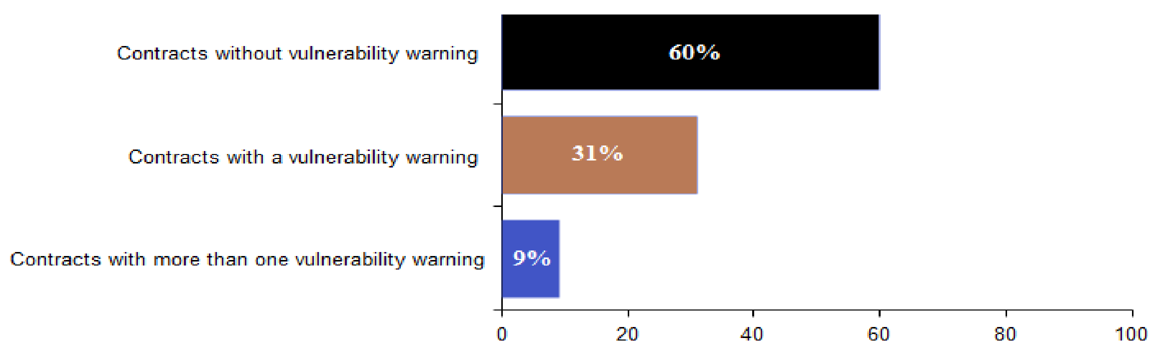

Figure 6. Comparison of smart contracts in terms of the number and type of vulnerability warning

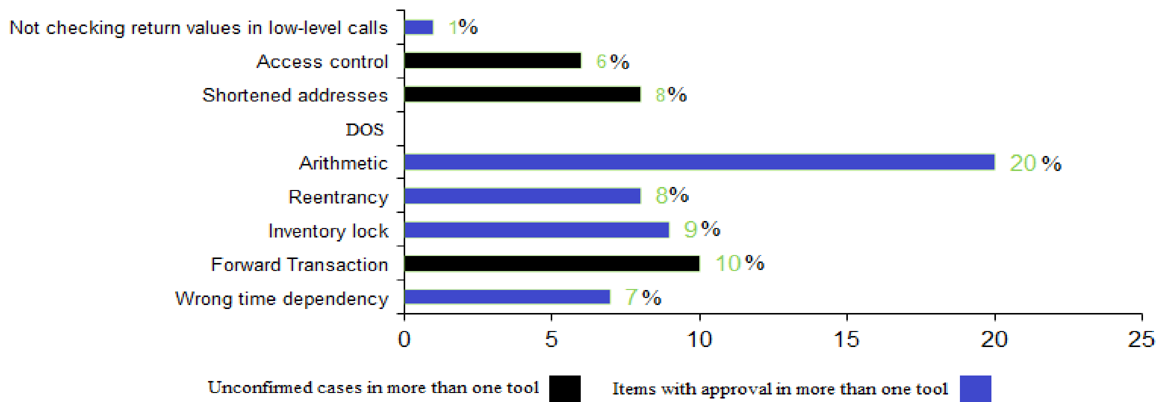

Figure 7. The results of checking the vulnerability alert in the dataset

In the chart below, you can see the mean, median and mode for alerts with confirmation in more than one tool.

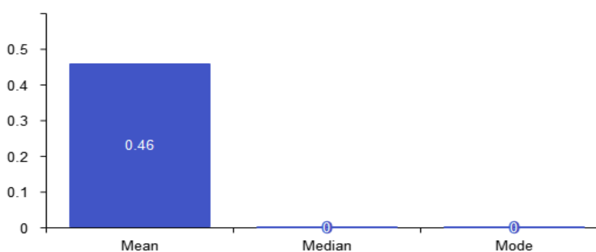

Figure 8. The mean, median and mode for alerts with confirmation in more than one tool

In Table 5, all tools that used in this section are compared.

Table 5. Comparison of automatic vulnerability detection tools

Tool name	Check method	The number of vulnerabilities to check from the list of 10 covered	Ability to access tool codes	Dynamic checking coverage
Slither	Intermediate Representation	8	yes (open source)	NO
Smartcheck	Intermediate Representation	6	yes (open source)	NO
Oyente	Symbolic Execution	4	yes (open source)	NO
Mythril	Symbolic Execution	8	yes (open source)	Yes

6. Conclusion and future work

Users can access the distributed infrastructure network through smart contracts in the blockchain sector. These agreements are crucial for information security since, in addition to their numerous uses, they directly impact the user's financial resources. Therefore, disclosing these contracts' vulnerabilities can be very helpful to blockchain application developers. In this study, the backdrop of the research and an explanation of a few technical words used in the field of blockchain technology were covered, along with an introduction to the issue and its significance.

In this study, the current vulnerabilities were thoroughly described, with real-world examples and practical solutions that were provided to fix them. Additionally, a statistical analysis of 100 active Ethereum smart contracts was done using the average results from four static security analysis tools. The Investigation's findings highlight the significance of the problem while also showing that several security weaknesses like arithmetic problem in the smart contracts under consideration recur. This represents a common response from developers to smart contract vulnerabilities. One of the primary causes of these problems in smart contracts may be the high complexity of large smart contracts and the developers' unfamiliarity with standard practices. With more contracts being evaluated and more (or more accurate and powerful) technologies being used, it is evident that the results will be more accurate. As a result, in further works based on this research, the number of vulnerabilities that have been identified would be raised, and testing can be carried out on a more considerable number of smart contracts. In addition, new automatic tools with various solutions are constantly being presented due to the spirit of active communities in this field. Future work should examine these tools' capabilities to finally identify why smart contract developers fail to pay attention to some widespread vulnerabilities. Better

training alternatives or ways to make smart contract programs less complicated for developers might be investigated further.

References

[1] Zou W, Lo D, Kochhar PS, Le XB, Xia X, Feng Y, Chen Z, Xu B. Smart contract development: Challenges and opportunities. IEEE Transactions on Software Engineering. 2019 Sep 24;47(10):2084-106.

[2] Wang S, Yuan Y, Wang X, Li J, Qin R, Wang FY. An overview of smart contract: architecture, applications, and future trends. In2018 IEEE Intelligent Vehicles Symposium (IV) 2018 Jun 26 (pp. 108-113). IEEE.

[3] Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., & Lee, H. N. (2022). Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, *10*, 6605-6621.

[4] Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., & Hierons, R. (2018, March). Smart contracts vulnerabilities: a call for blockchain software engineering?. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)* (pp. 19-25). IEEE.

[5] Sharma T, Zhou Z, Miller A, Wang Y. Exploring security practices of smart contract developers. arXiv preprint arXiv:2204.11193. 2022 Apr 24.

[6] Sifra EM. Security vulnerabilities and countermeasures of smart contracts: A survey. In2022 IEEE International Conference on Blockchain (Blockchain) 2022 Aug 22 (pp. 512-515). IEEE.

[7] Qian P, Liu Z, He Q, Huang B, Tian D, Wang X. Smart contract vulnerability detection technique: A survey. arXiv preprint arXiv:2209.05872. 2022 Sep 13.

[8] Singh, A., Parizi, R. M., Zhang, Q., Choo, K. K. R., & Dehghantanha, A. (2020). Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, *88*, 101654.

[9] Praitheeshan P, Pan L, Yu J, Liu J, Doss R. Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605. 2019 Aug 22.

[10] Wang Z, Jin H, Dai W, Choo KK, Zou D. Ethereum smart contract security research: survey and future research opportunities. Frontiers of Computer Science. 2021 Apr;15:1-8.

[11] Wohrer M, Zdun U. Smart contracts: security patterns in the ethereum ecosystem and solidity. In2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) 2018 Mar 20 (pp. 2-8). IEEE.

[12] Zheng Z, Zhang N, Su J, Zhong Z, Ye M, Chen J. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. arXiv preprint arXiv:2303.13770. 2023 Mar 24.

[13] Chen J, Huang M, Lin Z, Zheng P, Zheng Z. To healthier ethereum: A comprehensive and iterative smart contract weakness enumeration. arXiv preprint arXiv:2308.10227. 2023 Aug 20.

[14] Ray I. Security vulnerabilities in smart contracts as specifications in linear temporal logic (Master's thesis, University of Waterloo).

[15] He D, Deng Z, Zhang Y, Chan S, Cheng Y, Guizani N. Smart contract vulnerability analysis and security audit. IEEE Network. 2020 Jul 17;34(5):276-82.

[16] Yashavant CS, Kumar S, Karkare A. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. arXiv preprint arXiv:2202.11409. 2022 Feb 23.

[17] Ren M, Yin Z, Ma F, Xu Z, Jiang Y, Sun C, Li H, Cai Y. Empirical evaluation of smart contract testing: What is the best choice?. InProceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis 2021 Jul 11 (pp. 566-579).

[18] Zhou H, Milani Fard A, Makanju A. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. Journal of Cybersecurity and Privacy. 2022 May 27;2(2):358-78.

[19] Prasad B. Vulnerabilities and attacks on smart contracts over blockChain. Turkish Journal of Computer and Mathematics Education (TURCOMAT). 2021 May 10;12(11):5436-49.

[20] Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T. Defining smart contract defects on ethereum. IEEE Transactions on Software Engineering. 2020 Apr 20;48(1):327-45.

[21] Vani S, Doshi M, Nanavati A, Kundu A. Vulnerability Analysis of Smart Contracts. arXiv preprint arXiv:2212.07387. 2022 Dec 14.

[22] Durieux T, Ferreira JF, Abreu R, Cruz P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. InProceedings of the ACM/IEEE 42nd International conference on software engineering 2020 Jun 27 (pp. 530-541).

[23] Kushwaha SS, Joshi S, Singh D, Kaur M, Lee HN. Ethereum smart contract analysis tools: A systematic review. IEEE Access. 2022 Apr 22;10:57037-62.

[24] Wohrer M, Zdun U. Smart contracts: security patterns in the ethereum ecosystem and solidity. In2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) 2018 Mar 20 (pp. 2-8). IEEE.

[25] Zheng Z, Xie S, Dai HN, Chen X, Wang H. Blockchain challenges and opportunities: A survey. International journal of web and grid services. 2018;14(4):352-75.

[26] Lashkari B, Musilek P. A comprehensive review of blockchain consensus mechanisms. IEEE Access. 2021 Mar 12;9:43620-52.

[27] Zheng Z, Xie S, Dai H, Chen X, Wang H. An overview of blockchain technology: Architecture, consensus, and future trends. In2017 IEEE international congress on big data (BigData congress) 2017 Jun 25 (pp. 557-564). Ieee.

[28] Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Capkun S. On the security and performance of proof of work blockchains. InProceedings of the 2016 ACM SIGSAC conference on computer and communications security 2016 Oct 24 (pp. 3-16).

[29] Bentov I, Lee C, Mizrahi A, Rosenfeld M. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. ACM SIGMETRICS Performance Evaluation Review. 2014 Dec 8;42(3):34-7.

[30] Joshi S. Feasibility of proof of authority as a consensus protocol model. arXiv preprint arXiv:2109.02480. 2021 Aug 30.

[31] Ekparinya P, Gramoli V, Jourjon G. The attack of the clones against proof-of-authority. arXiv preprint arXiv:1902.10244. 2019 Feb 26.

[32] Manolache MA, Manolache S, Tapus N. Decision making using the blockchain proof of authority consensus. Procedia Computer Science. 2022 Jan 1;199:580-8.

[33] Singh PK, Singh R, Nandi SK, Nandi S. Managing smart home appliances with proof of authority and blockchain. InInnovations for Community Services: 19th International

Conference, I4CS 2019, Wolfsburg, Germany, June 24-26, 2019, Proceedings 19 2019 (pp. 221-232). Springer International Publishing.
[34] Saad SM, Radzi RZ. Comparative review of the blockchain consensus algorithm between proof of stake (pos) and delegated proof of stake (dpos). International Journal of Innovative Computing. 2020 Nov 19;10(2).
[35] Yang F, Zhou W, Wu Q, Long R, Xiong NN, Zhou M. Delegated proof of stake with downgrade: A secure and efficient blockchain consensus algorithm with downgrade mechanism. IEEE Access. 2019 Aug 14;7:118541-55.
[36] Hu Q, Yan B, Han Y, Yu J. An improved delegated proof of stake consensus algorithm. Procedia Computer Science. 2021 Jan 1;187:341-6.
[37] Snider M, Samani K, Jain T. Delegated proof of stake: features & tradeoffs. Multicoin Cap. 2018 Mar 2;19:1-9.
[38] Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS). 2002 Nov 1;20(4):398-461.
[39] Abraham I, Gueta G, Malkhi D, Alvisi L, Kotla R, Martin JP. Revisiting fast practical byzantine fault tolerance. arXiv preprint arXiv:1712.01367. 2017 Dec 4.
[40] Gao S, Yu T, Zhu J, Cai W. T-PBFT: An EigenTrust-based practical Byzantine fault tolerance consensus algorithm. China Communications. 2019 Dec;16(12):111-23.
[41] Consensus Algorithms in Blockchain Systems [Internet]. DEV Community. 2020 [cited 2024 Feb 3]. Available from: https://dev.to/akroutihamza/consensus-algorithms-in-blockchain-systems-44ag
[42] Makhdoom I, Abolhasan M, Ni W. Blockchain for IoT: The challenges and a way forward. InICETE 2018-Proceedings of the 15th International Joint Conference on e-Business and Telecommunications 2018 Jan 1.
[43] Lang D, Friesen M, Ehrlich M, Wisniewski L, Jasperneite J. Pursuing the vision of Industrie 4.0: Secure plug-and-produce by means of the asset administration shell and blockchain technology. In2018 IEEE 16th International Conference on Industrial Informatics (INDIN) 2018 Jul 18 (pp. 1092-1097). IEEE.
[44] 1.The Blockchain Generations [Internet]. Ledger. Available from: https://www.ledger.com/academy/blockchain/web-3-the-three-blockchain-generations
[45] Anwar S, Anayat S, Butt S, Butt S, Saad M. Generation Analysis of Blockchain Technology: Bitcoin and Ethereum. International Journal of Information Engineering & Electronic Business. 2020 Aug 1;12(4).
[46] Efanov D, Roschin P. The all-pervasiveness of the blockchain technology. Procedia computer science. 2018 Jan 1;123:116-21.
[47] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. Decentralized business review. 2008 Oct 31.
[48] Rostami M, Bahaghighat M, Zanjireh MM. Bitcoin daily close price prediction using optimized grid search method. Acta Universitatis Sapientiae, Informatica. 2021;13(2):265-87.
[49] Brighente A, Conti M, Kumar S. Extorsionware: Exploiting smart contract vulnerabilities for fun and profit. arXiv preprint arXiv:2203.09843. 2022 Mar 18.
[50] Egbertsen W, Hardeman G, van den Hoven M, van der Kolk G, van Rijsewijk A. Replacing paper contracts with Ethereum smart contracts. Semantic Scholar. 2016 Jun 10;35:1-35.
[51] Top Smart Contract Applications and Use Cases - Scalable Solutions [Internet]. 2021. Available from: https://scalablesolutions.io/news/smart-contract-applications-and-use-cases/
[52] Dannen C. Introducing Ethereum and solidity. Berkeley: Apress; 2017.
[53] Zhang P, Xiao F, Luo X. A framework and dataset for bugs in ethereum smart contracts. In2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) 2020 Sep 28 (pp. 139-150). IEEE.
[54] Krsul I, Spafford E, Tripunitara M. Computer vulnerability analysis. COAST Laboratory, Purdue University, West Lafayette, IN, Technical Report. 1998 May 6.
[55] Samreen NF, Alalfi MH. Reentrancy vulnerability identification in ethereum smart contracts. In2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE) 2020 Feb 18 (pp. 22-29). IEEE.
[56] Mehar MI, Shier CL, Giambattista A, Gong E, Fletcher G, Sanayhie R, Kim HM, Laskowski M. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. Journal of Cases on Information Technology (JCIT). 2019 Jan 1;21(1):19-32.
[57] Grossman S, Abraham I, Golan-Gueta G, Michalevsky Y, Rinetzky N, Sagiv M, Zohar Y. Online detection of effectively callback free objects with applications to smart contracts. Proceedings of the ACM on Programming Languages. 2017 Dec 27;2(POPL):1-28.
[58] Prechtel D, Groß T, Müller T. Evaluating spread of 'gasless send'in ethereum smart contracts. In2019 10th IFIP international conference on new technologies, mobility and security (NTMS) 2019 Jun 24 (pp. 1-6). IEEE.
[59] Oualid Z, Oualid Z. What is a reentrancy attack in Solidity? | Technical examples [Internet]. Get Secure World. 2022. Available from: https://www.getsecureworld.com/blog/what-is-a-reentrancy-attack-in-solidity-technical-examples/
[60] Samreen NF, Alalfi MH. A survey of security vulnerabilities in ethereum smart contracts. arXiv preprint arXiv:2105.06974. 2021 May 14.
[61] Samreen NF, Alalfi MH. A survey of security vulnerabilities in ethereum smart contracts. arXiv preprint arXiv:2105.06974. 2021 May 14.
[62] Palladino S. The parity wallet hack explained. July-2017.[Online]. Available: https://blog. zeppelin. solutions/on-the-parity-wallet-multisighack-405a8c12e8f7. 2017 Jul 20.
[63] Wöhrer M, Zdun U. Design patterns for smart contracts in the ethereum ecosystem. In2018 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 Jul 30 (pp. 1513-1520). IEEE.
[64] DASP - TOP 10 [Internet]. www.dasp.co. [cited 2024 Feb 3]. Available from: https://www.dasp.co
[65] Khan ZA, Namin AS. Ethereum smart contracts: Vulnerabilities and their classifications. In2020 IEEE International Conference on Big Data (Big Data) 2020 Dec 10 (pp. 1-10). IEEE.
[66] Thanh LY. Prevent Integer Overflow in Ethereum Smart Contracts [Internet]. Medium. 2018 [cited 2024 Feb 3]. Available from: https://yenthanh.medium.com/prevent-integer-overflow-in-ethereum-smart-contracts-a7c84c30de66
[67] Gao J, Liu H, Liu C, Li Q, Guan Z, Chen Z. Easyflow: Keep ethereum away from overflow. In2019 IEEE/ACM 41st International Conference on Software Engineering:

Companion Proceedings (ICSE-Companion) 2019 May 25 (pp. 23-26). IEEE.
[68] Scanning Live Ethereum Contracts for the “Unchecked-Send” Bug [Internet]. Hacking Distributed. Available from: https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/
[69] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok). InPrinciples of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6 2017 (pp. 164-186). Springer Berlin Heidelberg.
[70] Kulkarni Y. Denial of Service (DoS) Attack on Smart Contracts [Internet]. Be on the Right Side of Change. 2022. Available from: https://blog.finxter.com/denial-of-service-dos-attack-on-smart-contracts/
[71] Bhardwaj A, Shah SB, Shankar A, Alazab M, Kumar M, Gadekallu TR. Penetration testing framework for smart contract blockchain. Peer-to-Peer Networking and Applications. 2021 Sep;14:2635-50.
[72] Smart Contract Randomness or ReplicatedLogic Attack – Be on the Right Side of Change [Internet]. 2023 [cited 2024 Feb 3]. Available from: https://blog.finxter.com/randomness-or-replicatedlogic-attack-on-smart-contracts/
[73] Yao S, Zhang D. An Anonymous Verifiable Random Function with Applications in Blockchain. Wireless Communications and Mobile Computing. 2022 Apr 19;2022.
[74] Verifiable Random Function (VRF) - Explained | Chainlink [Internet]. chain.link. [cited 2024 Feb 3]. Available from: https://blog.chain.link/verifiable-random-function-vrf/
[75] Behnke R. What Is a Front-Running Attack? [Internet]. www.halborn.com. 2021 [cited 2024 Feb 3]. Available from: https://halborn.com/what-is-a-front-running-attack/
[76] Frontrunning - Ethereum Smart Contract Best Practices [Internet]. consensys.github.io. Available from: https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning/
[77] Mense A, Flatscher M. Security vulnerabilities in ethereum smart contracts. InProceedings of the 20th international conference on information integration and web-based applications & services 2018 Nov 19 (pp. 375-380).
[78] ImmuneBytes. A Techno-Manual on the Front Running Attack - ImmuneBytes [Internet]. 2022 [cited 2024 Feb 3]. Available from: https://www.immunebytes.com/blog/front-running-attack/
[79] Front-running attack in DeFi applications - how to deal with it? [Internet]. Securing. 2022. Available from: https://www.securing.pl/en/front-running-attack-in-defi-applications-how-to-deal-with-it/
[80] Libsubmarine.org. 2022. Available from: https://libsubmarine.org/
[81] Arulprakash M, Jebakumar R. Commit-reveal strategy to increase the transaction confidentiality in order to counter the issue of front running in blockchain. InAIP Conference Proceedings 2022 Aug 26 (Vol. 2460, No. 1). AIP Publishing.
[82] Dika A, Nowostawski M. Security vulnerabilities in ethereum smart contracts. In2018 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 Jul 30 (pp. 955-962). IEEE.
[83] Tang X, Zhou K, Cheng J, Li H, Yuan Y. The vulnerabilities in smart contracts: A survey. InAdvances in Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19-23, 2021, Proceedings, Part III 7 2021 (pp. 177-190). Springer International Publishing.
[84] Ethereum Contract Diff Checker [Internet]. etherscan.io. [cited 2024 Feb 3]. Available from: https://etherscan.io/contractdiffchecker?a1=0xa11e4ed59dc94e69612f3111942626ed513cb172
[85] Zhu H, Niu W, Liao X, Zhang X, Wang X, Li B, He Z. Attacker Traceability on Ethereum through Graph Analysis. Security and Communication Networks. 2022 Jan 27;2022.
[86] CoinFabrik. Smart Contract Short Address Attack Mitigation Failure [Internet]. CoinFabrik. 2017 [cited 2024 Feb 3]. Available from: https://blog.coinfabrik.com/smart-contract-short-address-attack-mitigation-failure/
[87] Perez D, Livshits B. Smart contract vulnerabilities: Vulnerable does not imply exploited. In30th USENIX Security Symposium (USENIX Security 21) 2021 (pp. 1325-1341).
[88] Perez D, Livshits B. Smart contract vulnerabilities: Vulnerable does not imply exploited. In30th USENIX Security Symposium (USENIX Security 21) 2021 (pp. 1325-1341).
[89] Sayeed S, Marco-Gisbert H, Caira T. Smart contract: Attacks and protections. IEEE Access. 2020 Jan 30;8:24416-27.
[90] Bug Security : Locked Ether · Issue #19930 · ethereum/go-ethereum [Internet]. GitHub. [cited 2024 Feb 3]. Available from: https://github.com/ethereum/go-ethereum/issues/19930
[91] Smart Contract Weakness Classification (SWC) [Internet]. swcregistry.io. [cited 2024 Feb 3]. Available from: https://swcregistry.io
[92] SmartCDS/Addresses.txt at main · Csreasercher/SmartCDS [Internet]. GitHub. [cited 2024 Feb 3]. Available from: https://github.com/Csreasercher/SmartCDS/blob/main/Addresses.txt
[93] Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) 2019 May 27 (pp. 8-15). IEEE.
[94] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. Smartcheck: Static analysis of ethereum smart contracts. InProceedings of the 1st international workshop on emerging trends in software engineering for blockchain 2018 May 27 (pp. 9-16).
[95] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. InProceedings of the 2016 ACM SIGSAC conference on computer and communications security 2016 Oct 24 (pp. 254-269).
[96] MythX: Preparing for a smart contract audit [Internet]. mythx.io. [cited 2024 Feb 3]. Available from: https://mythx.io/about
[97] Chen T, Cao R, Li T, Luo X, Gu G, Zhang Y, Liao Z, Zhu H, Chen G, He Z, Tang Y. SODA: A Generic Online Detection Framework for Smart Contracts. InNDSS 2020 Feb 23.
[98] Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. InProceedings of the ACM/IEEE 42nd International Conference on Software Engineering 2020 Jun 27 (pp. 778-788).

[99] Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. IEEE Transactions on Software Engineering. 2021 Jan 27;48(7):2189-207.