**Ethereum smart contract research: Survey shortcomings and propose improvements**

**LUO Shuhai1 , Shen Linling1, ZHU Jun2 , ZHAO Yuhao2, LIN Lixin2**

1 Department of Data Science, Lingnan University, Tuen Mun district of New Territories, Hong Kong

2 Department of Artificial Intelligence and Business Analytics, Lingnan University, Tuen Mun district of New Territories, Hong Kong

**Abstract**

Ethereum is one of the most important parts of blockchain technology, the core of it is smart contracts. In recent years, Ethereum's smart contract ecosystem has experienced several major security events in its development, exposing vulnerabilities and design flaws in smart contracts and driving improvements in contract development and management. We propose to find a new technology to reduce vulnerabilities in smart contracts and decrease the risk of incurring losses.

**Keyword: Blockchain, Ethereum, Smart Contract, Security, Vulnerabilities, Design Flaw**

1. **Introduction**
   1. **Ethereum Background**

Ethereum is an open-source public blockchain platform with smart contract functions. In essence, Ethereum is a public database that keeps permanent records of digital transactions. The operation of Ethereum is based on blockchain technology, a decentralized and decentralized distributed ledger technology[1], which consists of a series of chronological data blocks, each containing a certain number of transaction records and the hash value of the previous data block to form a chain structure. This chain structure makes data impossible to tamper with, not changed or deleted if it is added to the blockchain.

Compared with the bitcoin, the working principle of Ethereum is similar, but the function of Ethereum is more flexible in the Ethereum work process. At first, Users send transactions or deploy smart contracts via their wallets, then Transactions and smart contracts are validated by the nodes in the network and executed in the Ethereum Virtual Machine (EVM), verifying the execution of the smart contract. The validated transaction and contract results are packaged into blocks and added to the blockchain[2].

Ethereum allows developers to write smart contracts and deploy them on the blockchain. Arguably we can say smart contracts are essentially a core component of Ethereum. Smart contract use programming language to convert the contract terms into a program that can be run. After the smart contract program starts to be executed on the blockchain, it will leave permanent records on the blockchain, and the records cannot be tampered with which makes the transaction process transparent and open, ensuring the legality of the contract execution. The encrypted execution of smart contracts in Ethereum relies on the Ethereum Virtual Machine (EVM)[3]. The EVM is the core of the Ethereum network and is responsible for executing the smart contract code. All nodes will run the EVM and verify the execution results of the contract. This distributed execution method ensures the reliability and security of smart contracts, as any tampering with the contract will be discovered and rejected by other nodes. Smart contracts do not need third-party intermediaries. When certain conditions are met, they operate automatically and execute completely in accordance with the programs written in the Solidity language, which can prevent fraud and third-party interference. Thus, transaction costs and risks can be reduced[4].

* 1. **Hash functions in Ethereum**

Hash functions play an important role in Ether, where transactions and blocks are encrypted and verified using hash algorithms. Hash function is an algorithm that creates a fixed-length and unique hash value from a message of arbitrary length. Hash algorithm has three characteristics: uniqueness (collision resistance), unidirectional performance, and avalanche effect. Uniqueness means that the hash value of any information is unique and the same hash values do not appear. Unidirectional performance means that it is extremely difficult to launch the original information according to the hash value to ensure the security of the information. The avalanche effect means that the hash value changes dramatically even if the information changes minimally[5].

* 1. **Hash Algorithms**

Currently, two hash algorithms are mainly used in Ethereum:SHA-256 and Keccak-256,to ensure the security of digital currency production[5]. The SHA-256 algorithm inputs the data and goes through a series of complex logical and mathematical operations to eventually output a 256-bit hash value.

In Ethereum, the more widely used is Keccak-256.Keccak-256 is an encryption function that is part of the SHA-3 series. It uses a unique sponge structure and consists of an absorption phase and a squeezing phase. In the absorption phase, the input message is divided into chunks and processed by a substitution function; in the squeezing phase, the output is extracted from the state by repeatedly applying the same substitution function[6]. The Keccak-256 is widely used in Ethereum, including address generation, smart contracts, mining, and blockchain security. In address generation, the user's wallet address is generated by hash processing of its public key using Keccak-256.

In smart contracts, Keccak-256 is used to validate digital signatures, generate random numbers, and ensure the integrity of the data. In mining, Keccak-256 is used in Ethereum's Proof of Workload (PoW) algorithm, where miners need to solve a complex mathematical problem involving Keccak-256 hash operations to compete for accounting rights[7].

* 1. **Ethereum technical bottlenecks**

Although Ethereum is one of the main forces of blockchain technology, with the continuous development of blockchain technology and the continuous upgrading of user needs, Ethereum is still facing technical bottlenecks, mainly reflected in the following aspects:

1. Restricted extensibility

Throughput bottleneck: The Ethereum network has a limited throughput, i. e., a limited number of transactions that can be processed in a unit of time. This results in periods of high demand, such as the NFT boom and the rise of DeFi apps, where networks are prone to congestion, surging transaction costs, and declining user experience[8].

1. Delay bottleneck

Delay is the time elapsed from submitting the transaction to the time when the transaction is confirmed and included in the blockchain. The delay problem of the Ethereum network also affects its scalability. Longer confirmation time and blockchain synchronization time increase the waiting time for transactions and reduce the efficiency of the network[9].

1. Smart contract security

Smart contracts are one of the core features of Ethereum that allows users to execute automated contracts on the blockchain. However, the security issue of smart contracts is also one of the technical bottlenecks facing Ethereum. Since the code of smart contracts is difficult to modify once deployed, any security breach can lead to serious consequences, such as fund theft and data leakage. In addition, the complexity of smart contracts also increases the risk of security vulnerabilities.

1. Consensus mechanism

Ethereum currently uses a consensus agreement of proof of work (Proof of Work, PoW) and proof of equity (Proof of Stake, PoS), but it is still mainly PoW. This consensus protocol has a high guarantee of security, but there are also some problems. For example, the PoW consensus agreement could lead to network congestion and higher transaction costs. It also requires a lot of computing resources and power consumption, which is detrimental to environmental protection and sustainable development.

1. Privacy

Ethereum's transaction data is open and transparent, which helps to increase the transparency and traceability of transactions. However, it also raises the risk of privacy leakage. Some sensitive information, such as user identity, transaction amount, and so on, may be used by malicious attackers for illegal activities.

* 1. **Research directions for breakthrough**

For the current defects of Ethereum, there are three directions that can be used as future breakthrough points:

1. The Proof of Interest (PoS) mechanism

We can turn proof of work (PoW) to proof of equity (PoS), to reduce energy consumption and increase transaction speed. However, how to ensure that the network remains highly security and operates steady under the PoS mechanism and keeps the degree of decentralization of the network, it still needs to be supported by newer technological breakthroughs[10].

1. Blockchain sharding technology

To improve the throughput and scalability of the Ethereum network, we can try sharding technology, which divides the blockchain network into multiple smaller parts. Each part of blockchain can deal with transactions and store data independently. But it will face a lot of problem to realize the sharding technology, such as how to ensure that the communication and data are identical between parts[11].

1. Layer 2

Layer 2 solutions are an additional network layer built on top of the main Ethernet network, in order to deal with more transactions and reduce the burden of the main net. We can improve the scalability and performance of Ethereum by optimize the transaction processing process and improve the transaction efficiency[12].

1. **Potential Issues and Solutions**
   1. **Security Vulnerabilities and Attacks**
2. Reentrancy Attacks

One of the most infamous vulnerabilities is the reentrancy attack, exemplified by the DAO hack. In such attacks, a malicious contract repeatedly calls back into the vulnerable contract before the initial function completes, leading to the depletion of funds.

**Solution** Implementing the Checks-Effects-Interactions pattern, utilizing reentrancy guards like ReentrancyGuard from OpenZeppelin, and thoroughly auditing contracts can mitigate this risk.

1. Integer Overflow and Underflow

These occur when arithmetic operations exceed the maximum or minimum values a variable can hold.

**Solution** Using the SafeMath library, which is widely adopted in the Ethereum community, helps prevent these issues by ensuring safe arithmetic operations.

1. Phishing and Social Engineering

Users can be tricked into interacting with malicious contracts or websites that appear legitimate.

**Solution** Educating users about the risks of phishing, promoting secure practices such as double-checking URLs, and using tools that warn users about unverified contracts can help reduce these risks.

1. Front-running

This occurs when an attacker observes a pending transaction and places a similar transaction with higher gas fees to be mined first.

**Solution** Techniques like commit-reveal schemes and adding randomness to transaction data can help prevent front-running. Layer 2 solutions and other off-chain mechanisms can also offer protection.

* 1. **Scalability Issues**

As the Ethereum network grows, scalability becomes a pressing concern. High demand can lead to network congestion, driving up gas fees and making transactions slow and expensive.

**Solution** The Ethereum 2.0 upgrade aims to address scalability through the introduction of sharding and a transition from Proof of Work (PoW) to Proof of Stake (PoS). Layer 2 solutions like Optimistic Rollups and zk-Rollups also promise to enhance scalability by processing transactions off-chain and settling them on-chain.

* 1. **Usability Challenges**

1. Complexity and User Experience

Interacting with smart contracts can be complex and intimidating for non-technical users.

**Solution** Improving the user interface and user experience (UI/UX) of decentralized applications (dApps), providing clear documentation, and offering user-friendly wallets and tools can enhance usability. Projects like MetaMask have made significant strides in making Ethereum more accessible.

1. Gas Fees

Fluctuating gas prices can make it difficult for users to predict costs, leading to either overpayment or failed transactions due to underpayment.

**Solution** Implementing gas optimization techniques within smart contracts and educating users on setting appropriate gas limits can help. EIP-1559, which was implemented in the London hard fork, introduces a base fee model to make gas fees more predictable.

* 1. **Regulatory and Legal Risks**

Smart contracts operate in a relatively new and evolving regulatory landscape. Legal uncertainties can pose risks for both developers and users.

**Solution** Staying informed about regulatory developments, engaging with legal experts, and considering compliance from the outset of contract development can help mitigate legal risks. Additionally, adopting decentralized governance models can distribute regulatory risk among a broader community.

* 1. **Code Immutability**

Once deployed, smart contracts on Ethereum are immutable, meaning they cannot be altered. This immutability can be a double-edged sword; while it ensures trustlessness and security, it also means that bugs and vulnerabilities cannot be easily fixed.

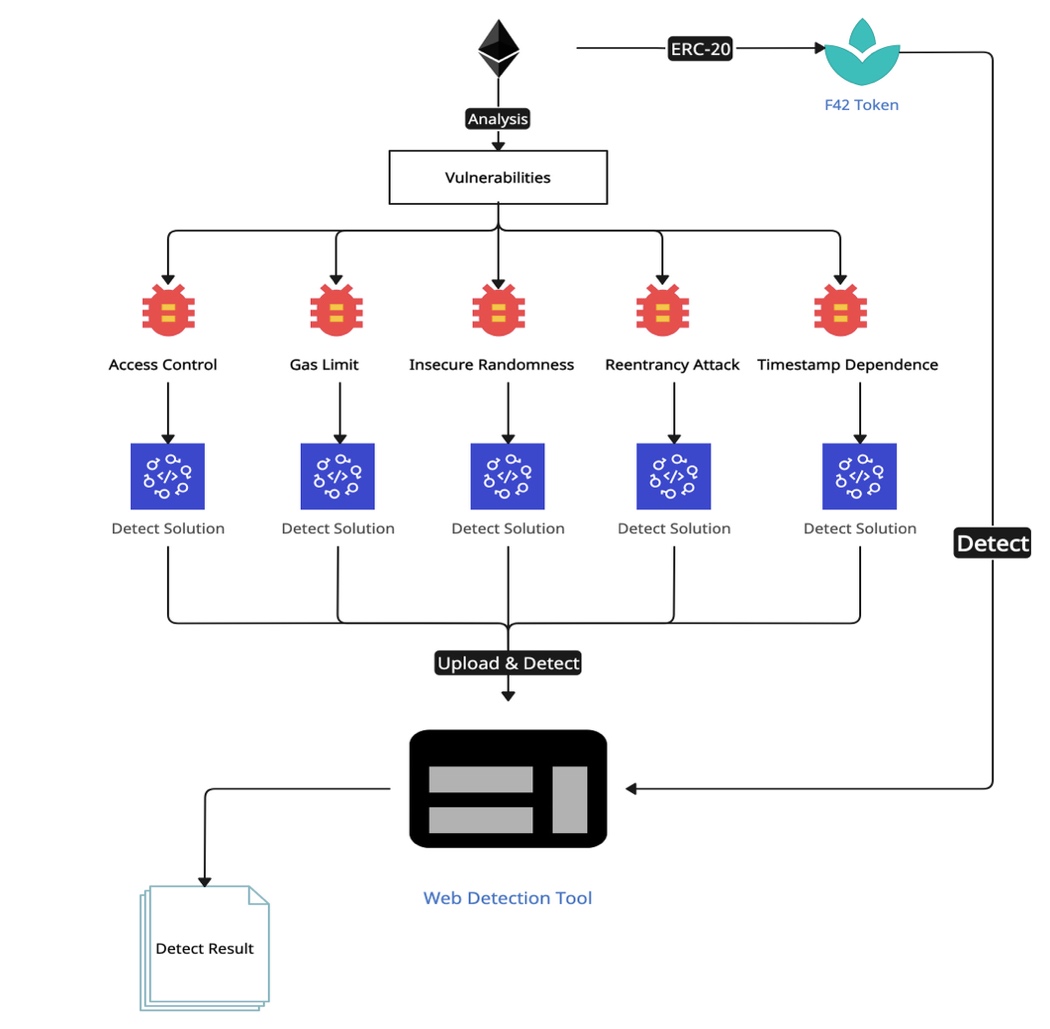
**Solution** Implementing upgradeable contract patterns, such as the Proxy pattern, allows developers to introduce new logic while maintaining the same contract address. Rigorous testing, peer reviews, and professional audits before deployment are also crucial.

* 1. **Oracles and External Data Dependency**

Smart contracts often rely on external data sources (oracles) to execute functions based on real-world events. If an oracle is compromised, it can lead to incorrect or malicious contract behavior.

**Solution** Using decentralized oracles like ChainLink can reduce the risk of a single point of failure. Ensuring that multiple oracles are used and implementing fallback mechanisms can provide additional security[12].

1. **Solution Implementation**
   1. **Overview of project work**

****

Picture 1: Project Workflow

Based on our analyses of the different vulnerabilities in Ether, we have decided to target each of the following vulnerabilities that we will be working on and design a methodology for detecting vulnerabilities based on the characteristics of each smart contract that has the vulnerability and create a cryptocurrency that is part of our project, which is a cryptocurrency that does not have the characteristics of the above-mentioned vulnerabilities, as a result of the improvements we made in relation to the vulnerabilities. Then we will design a corresponding code for detecting vulnerabilities for each classical vulnerability of Ether, and finally aggregate so the detection code and integrate to develop a website dedicated to detecting vulnerabilities of Ether smart contracts.

* 1. **Create cryptocurrency**

Develop a cryptocurrency belonging to our project according to the standards of ERC-20 and name it F42. Additionally, This is a cryptocurrency without the Ethereum vulnerability The details of the solidity code are shown below.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.28;

**import** "./IERC20.sol";

contract F42Token is IERC20 {

// Token basic info.

string **public** name = "F42Token";

string **public** symbol = "F42";

uint8 **public** decimals = 18;

uint256 **public** totalSupply;

// Record balance of certain address.

mapping(address => uint256) **public** balanceOf;

// Record limit of certain address.

mapping(address => mapping(address => uint256)) **public** allowance;

// Set total coin via user input.

// Give them to creater address.

constructor(uint256 \_initialSupply) {

totalSupply = \_initialSupply \* 10\*\*uint256(decimals);

balanceOf[msg.sender] = totalSupply;

emit Transfer(address(0), msg.sender, totalSupply);

}

// Implementation of interface.

function transfer(address \_to, uint256 \_value)

**public**

returns (bool success)

{

require(\_to != address(0), "Invalid address");

require(balanceOf[msg.sender] >= \_value, "Insufficient balance");

balanceOf[msg.sender] -= \_value;

balanceOf[\_to] += \_value;

emit Transfer(msg.sender, \_to, \_value);

**return** **true**;

}

// Implementation of interface.

function approve(address \_spender, uint256 \_value)

**public**

returns (bool success)

{

require(\_spender != address(0), "Invalid address");

allowance[msg.sender][\_spender] = \_value;

emit Approval(msg.sender, \_spender, \_value);

**return** **true**;

}

// Implementation of interface.

function transferFrom(

address \_from,

address \_to,

uint256 \_value

) **public** returns (bool success) {

require(\_to != address(0), "Invalid address");

require(balanceOf[\_from] >= \_value, "Insufficient balance");

require(allowance[\_from][msg.sender] >= \_value, "Allowance exceeded");

balanceOf[\_from] -= \_value;

balanceOf[\_to] += \_value;

allowance[\_from][msg.sender] -= \_value;

emit Transfer(\_from, \_to, \_value);

**return** **true**;

}

}

* 1. **Vulnerabilities and Corresponding Solutions**
     1. **Access Control**

1. **Vulnerable Contract**

This Solidity code defines a smart contract called Access Control Vulnerability. The contract allows users to make deposits and withdrawals, and to check the balance of funds within the contract. However, the ‘Access Control Vulnerability’ in the contract name implies an access control vulnerability.

Potential vulnerability: Lack of access control

Security issue: There is no access control on the withdraw() function. This means that anyone can call withdraw() and withdraw funds from a contract, even though they haven't deposited any funds. This is an access control vulnerability because even though there is an owner variable, it is not used or restricted in withdrawals.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract AccessControlVulnerability {

    address **public** owner;

    uint256 **public** funds;

    constructor() {

        owner = msg.sender;

    }

    function deposit() **public** payable {

        funds += msg.value;

    }

    function withdraw(uint256 amount) **public** {

        require(amount <= funds, "Insufficient funds");

        (bool success, ) = msg.sender.call{value: amount}("");

        require(success, "Transfer failed");

        funds -= amount;

    }

    function getFunds() **public** view returns (uint256) {

**return** funds;

    }

}

1. **Detection Code**

This Python code serves to check for access control vulnerabilities in smart contracts and uses the security analysis tool Slither for vulnerability detection. The details of the python code are shown below.

**import** os

**from** smart\_contract.vulnerabilities **import** check\_with\_slither, contract, result

**def** check\_access\_control\_with\_slither(result\_path) -> str:

**print**(f"Checking for access control vulnerabilities in {result\_path} with Slither...")

**try**:

        with open(result\_path, "r", encoding="utf-8") as f:

            content = f.read()

        start\_string = "AccessControlVulnerability"

        end\_string = "INFO"

        start = content.find(start\_string)

        end = -1

**if** start != -1:

            detection = content[start:]

            end = detection.find(end\_string)

**print**("Potential access control vulnerability found!")

**print**(detection[:end])

**return** detection[:end]

**else**:

**print**("No access control vulnerability found.")

**return** "No access control vulnerability found."

**except** Exception as e:

**print**(f"An error occurred while running Slither: {e}")

**if** \_\_name\_\_ == "\_\_main\_\_":

**if** **not** os.path.exists(result):

        check\_with\_slither(contract)

    check\_access\_control\_with\_slither(result)

* + 1. **Gas Limit**
  1. **Vulnerable Contract**

Gas Limit Vulnerability, its primary function is to allow users to deposit funds and ultimately distribute all funds in the contract equally to all users. The contract has a potential ‘Gas Limit’ vulnerability, which may cause the contract to fail to function properly under certain circumstances.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract GasLimitVulnerability {

    address **public** owner;

    mapping(address => uint256) **public** balances;

    address[] **public** users;

    constructor() {

        owner = msg.sender;

    }

**function** addUser(address user) **public** payable {

        require(msg.value > 0, "Must send some ether");

**if** (balances[user] == 0) {

            users.push(user);

        }

        balances[user] += msg.value;

    }

**function** distributeFunds() **public** {

        require(msg.sender == owner, "Only owner can distribute funds");

        uint256 amountPerUser = address(**this**).balance / users.length;

**for** (uint256 i = 0; i < users.length; i++) {

            (bool success, ) = users[i].call{value: amountPerUser}("");

            require(success, "Transfer failed");

        }

    }

**function** getUserCount() **public** view returns (uint256) {

**return** users.length;

    }

}

* 1. **Detection Code**

The main purpose of this Python code is to check Solidity smart contracts for vulnerabilities related to the Gas limit by calling Slither, a tool for analyzing the security of smart contracts and identifying vulnerabilities, especially in Solidity contracts.

**import** os

from smart\_contract.vulnerabilities **import** check\_with\_slither, contract, result

def check\_gas\_limit\_with\_slither(result\_path) -> str:

    print(f"Checking for gas limit vulnerabilities in {result\_path} with Slither...")

**try**:

**with** open(result\_path, "r", encoding="utf-8") as f:

            content = f.read()

        start\_string = "GasLimitVulnerability"

        end\_string = "INFO"

        start = content.find(start\_string)

        end = -1

**if** start != -1:

            detection = content[start:]

            end = detection.find(end\_string)

            print("Potential gas limit vulnerability found!")

            print(detection[:end])

**return** detection[:end]

**else**:

            print("No gas limit vulnerability found.")

**return** "No gas limit vulnerability found."

    except Exception as e:

        print(f"An error occurred while running Slither: {e}")

**if** \_\_name\_\_ == "\_\_main\_\_":

**if** not os.path.exists(result):

        check\_with\_slither(contract)

    check\_gas\_limit\_with\_slither(result)

* + 1. **Insecure Randomness**
  1. **Vulnerable Contract**

This Solidity smart contract defines a simple insecure random number generation logic for choosing a winner. The name of the contract, Unsecure Randomness, also implies that there are security issues with its random number generation mechanism.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract UnsecureRandomness {

    address **public** winner;

**function** pickWinner() **public** {

        uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender)));

**if** (random % 2 == 0) {

            winner = msg.sender;

        }

    }

**function** getWinner() **public** view returns (address) {

**return** winner;

    }

}

* 1. **Detection Code**

The function of this code is to check for insecure random number generation vulnerabilities (insecure randomness) in smart contracts by calling the Slither tool. Specifically, it analyses the analysis report file generated by Slither for vulnerabilities related to insecure randomness and outputs the results.

**import** os

from smart\_contract.vulnerabilities **import** check\_with\_slither, contract, result

def check\_insecure\_randomness\_with\_slither(result\_path) -> str:

    print(f"Checking for insecure randomness vulnerabilities in {result\_path} with Slither...")

**try**:

**with** open(result\_path, "r", encoding="utf-8") as f:

            content = f.read()

        start\_string = "UnsecureRandomness"

        end\_string = "INFO"

        start = content.find(start\_string)

        end = -1

**if** start != -1:

            detection = content[start:]

            end = detection.find(end\_string)

            print("Potential insecure randomness vulnerability found!")

            print(detection[:end])

**return** detection[:end]

**else**:

            print("No insecure randomness vulnerability found.")

**return** "No insecure randomness vulnerability found."

    except Exception as e:

        print(f"An error occurred while running Slither: {e}")

**if** \_\_name\_\_ == "\_\_main\_\_":

**if** not os.path.exists(result):

        check\_with\_slither(contract)

    check\_insecure\_randomness\_with\_slither(result)

* + 1. **Reentrancy Attacks**
  1. **Vulnerable Contract**

This smart contract allows users to make deposits and withdrawals and check their balance. Despite the simplicity of the functionality, it suffers from a serious security vulnerability: a re-entry attack, which is due to the fact that the transfer of funds occurs before the balance is updated.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract VulnerableContract {

    mapping(address => uint256) **public** balances;

**function** deposit() **public** payable {

        balances[msg.sender] += msg.value;

    }

**function** withdraw(uint256 amount) **public** {

        require(balances[msg.sender] >= amount, "Insufficient balance");

        (bool success, ) = msg.sender.call{value: amount}("");

        require(success, "Transfer failed");

        balances[msg.sender] -= amount;

    }

**function** getBalance() **public** view returns (uint256) {

**return** balances[msg.sender];

    }

}

* 1. **Detection Code**

The main function of this code is to analyse smart contracts through the Slither tool and check for re-entry attack vulnerabilities from the reports generated by Slither. First, check if the result file analysed by Slither exists. If it does not exist, call ‘check\_with\_slither(contract)’ to analyse the specified contract.

Once the result file exists, the ‘check\_reentrancy\_with\_slither(result)’ function is called to look for potential reentry attack vulnerabilities from the result file. If a vulnerability is detected, a detailed description of the vulnerability is output; if no vulnerability is detected, the appropriate prompt is output.

**import** os

from smart\_contract.vulnerabilities **import** check\_with\_slither, contract, result

def check\_reentrancy\_with\_slither(result\_path) -> str:

    print(f"Checking for reentrancy vulnerabilities in {result\_path} with Slither...")

**try**:

**with** open(result\_path, "r", encoding="utf-8") as f:

            content = f.read()

        start\_string = "Reentrancy in"

        end\_string = "INFO"

        start = content.find(start\_string)

        end = -1

**if** start != -1:

            detection = content[start:]

            end = detection.find(end\_string)

            print("Potential reentrancy vulnerability found!")

            print(detection[:end])

**return** detection[:end]

**else**:

            print("No reentrancy vulnerability found.")

**return** "No reentrancy vulnerability found."

    except Exception as e:

        print(f"An error occurred while running Slither: {e}")

**if** \_\_name\_\_ == "\_\_main\_\_":

**if** not os.path.exists(result):

        check\_with\_slither(contract)

    check\_reentrancy\_with\_slither(result)

* + 1. **Timestamp Dependence**
  1. **Vulnerable Contract**

This Solidity code defines a smart contract called Time stamp Dependency, which involves timestamp-based logic and therefore may be vulnerable to timestamp dependency.

Timestamp Dependency issue:

block.timestamp is the timestamp of the current block, which is set by the miner in each block. While miners are not completely free to choose the timestamp, they can manipulate it within a certain range.

In this contract, the choice of winner depends entirely on the even or odd nature of the timestamp. If the timestamp is even, the caller is set as the winner. Thus, a miner can influence the outcome by manipulating the timestamp to ensure that a particular user wins. For example, a miner can adjust the timestamp to make it even or odd when packaging a transaction to help a particular address become a winner.

Magnitude of the problem:

The timestamp dependency problem is a security vulnerability for logic involving random number generation or fair selection. Because miners can partially manipulate timestamps, it results in a game or selection process that is no longer fair and could be exploited maliciously.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract TimestampDependency {

    address **public** winner;

**function** play() **public** {

**if** (block.timestamp % 2 == 0) {

            winner = msg.sender;

        }

    }

**function** getWinner() **public** view returns (address) {

**return** winner;

    }

}

* 1. **Detection Code**

The main function of this code is to check for timestamp dependency vulnerabilities in smart contracts by analysing the results generated by the Slither tool.

**import** os

from smart\_contract.vulnerabilities **import** check\_with\_slither, contract, result

def check\_timestamp\_dependence\_with\_slither(result\_path) -> str:

    print(f"Checking for timestamp dependence vulnerabilities in {result\_path} with Slither...")

**try**:

**with** open(result\_path, "r", encoding="utf-8") as f:

            content = f.read()

        start\_string = "TimestampDependency"

        end\_string = "INFO"

        start = content.find(start\_string)

        end = -1

**if** start != -1:

            detection = content[start:]

            end = detection.find(end\_string)

            print("Potential timestamp dependence vulnerability found!")

            print(detection[:end])

**return** detection[:end]

**else**:

            print("No timestamp dependence vulnerability found.")

**return** "No timestamp dependence vulnerability found."

    except Exception as e:

        print(f"An error occurred while running Slither: {e}")

**if** \_\_name\_\_ == "\_\_main\_\_":

**if** not os.path.exists(result):

        check\_with\_slither(contract)

    check\_timestamp\_dependence\_with\_slither(result)

1. **Detection Implementation**

As mentioned above, we have successfully implemented the function of detecting vulnerabilities for all the vulnerabilities we want to study as well as developing smart contracts that do not have the characteristics of the vulnerabilities mentioned above, so next we consolidated all the detection methods and launched a website dedicated to detecting vulnerability detection tools, the specific details about this tool are as follows.

* 1. **Detection page design**

The following code creates an application based on the Flask web framework, whose main function is to accept smart contract code uploaded by users, use the Slither[13] tool to detect vulnerabilities in smart contracts, and display the results on the front-end page of the website. Specifically, it checks for several common smart contract security vulnerabilities and counts the number of vulnerabilities found.

from flask **import** Flask, render\_template, request

from smart\_contract.access\_control.access\_control **import** check\_access\_control\_with\_slither

from smart\_contract.gas\_limit.gas\_limit **import** check\_gas\_limit\_with\_slither

from smart\_contract.insecure\_randomness.insecure\_randomness **import** check\_insecure\_randomness\_with\_slither

from smart\_contract.reentrancy\_attacks.reentrancy\_attacks **import** check\_reentrancy\_with\_slither

from smart\_contract.timestamp\_dependence.timestamp\_dependence **import** check\_timestamp\_dependence\_with\_slither

from smart\_contract.vulnerabilities **import** check\_with\_slither

app = Flask(\_\_name\_\_)

contract = "./input.sol"

result = "./result.txt"

@app.route("/", methods=["GET", "POST"])

def home():

    detection = ""

    v\_num = 0

**if** request.method == "POST":

        user\_text = request.form.get("user\_text")

**if** user\_text:

            detection, v\_num = detect\_vulnerability(user\_text)

**return** render\_template("index.html", detection=detection, v\_num=v\_num)

def detect\_vulnerability(text) -> (str, **int**):

**with** open(contract, "w", encoding="utf-8") as file:

        file.write(text)

    check\_with\_slither(contract)

    v\_num = 0

    reentrancy = check\_reentrancy\_with\_slither(result)

    v\_num += count(reentrancy)

    timestamp\_dependence = check\_timestamp\_dependence\_with\_slither(result)

    v\_num += count(timestamp\_dependence)

    access\_control = check\_access\_control\_with\_slither(result)

    v\_num += count(access\_control)

    insecure\_randomness = check\_insecure\_randomness\_with\_slither(result)

    v\_num += count(insecure\_randomness)

    gas\_limit = check\_gas\_limit\_with\_slither(result)

    v\_num += count(gas\_limit)

**return** (f"reentrancy: {reentrancy}\n\n"

            f"timestamp dependence: {timestamp\_dependence}\n\n"

            f"access control: {access\_control}\n\n"

            f"insecure randomness: {insecure\_randomness}\n\n"

            f"gas limit: {gas\_limit}\n\n"), v\_num

def count(text: str) -> **int**:

**return** 0 **if** text.startswith("No") **else** 1

**if** \_\_name\_\_ == "\_\_main\_\_":

    app.run(host="0.0.0.0", port=8080, debug=True)

**The core code of detection design**

Above code detect\_vulnerability(text): this is the core vulnerability detection function that receives the smart contract code (text) uploaded by the user and performs the following steps:

**Save contract code** save the contract code uploaded by the user to the input.sol file.

**Static analysis** call check\_with\_slither(contract), use Slither tool to statically analyse the input.sol contract file, and save the result in result.txt file.

**Vulnerability checking** call each specific vulnerability detection function and analyse the results from the result.txt file

**check\_reentrancy\_with\_slither(result)** check for reentry attack vulnerabilities.

**check\_timestamp\_dependence\_with\_slither(result)** checks for timestamp dependence vulnerabilities.

**check\_access\_control\_with\_slither(result)** check for access control vulnerabilities.

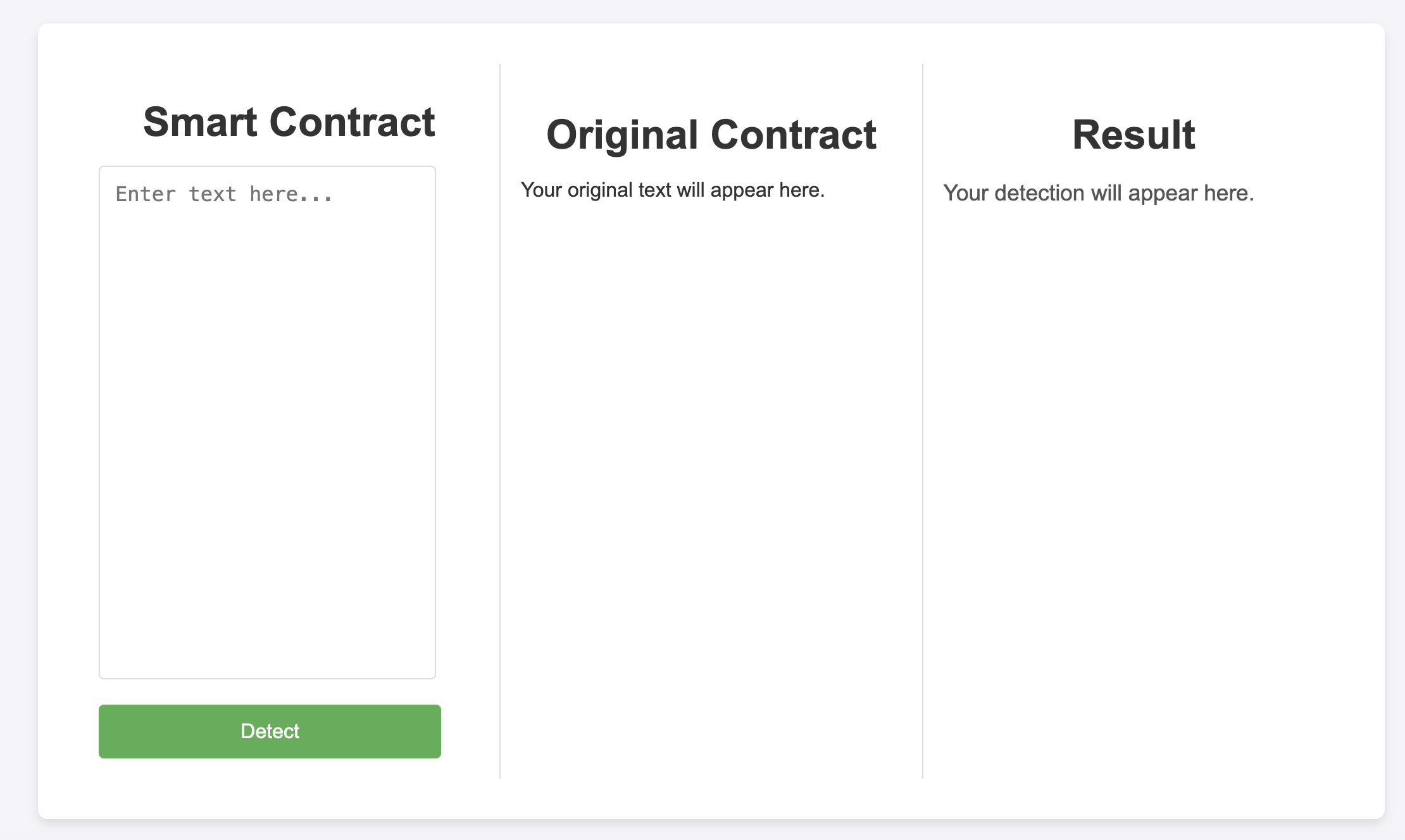
**check\_insecure\_randomness\_with\_slither(result)** check for insecure random number generation.

**check\_gas\_limit\_with\_slither(result)** check for gas limit issues.

**Vulnerability count** call count() function to count the number of each vulnerability.

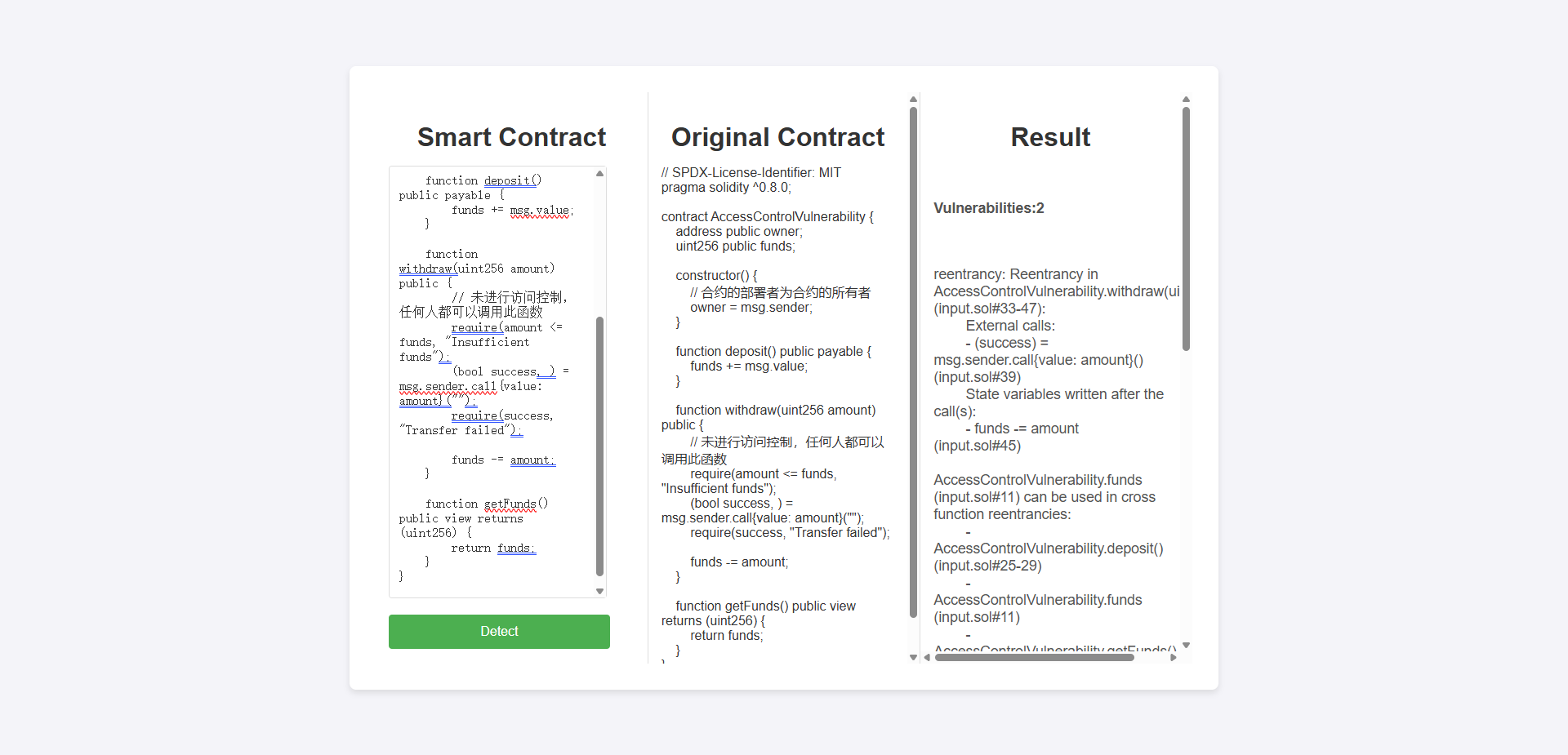
**Return test result** return a tuple containing the test result string and the total number of vulnerabilities.

The picture below is our dedicated page for contract vulnerability detection.



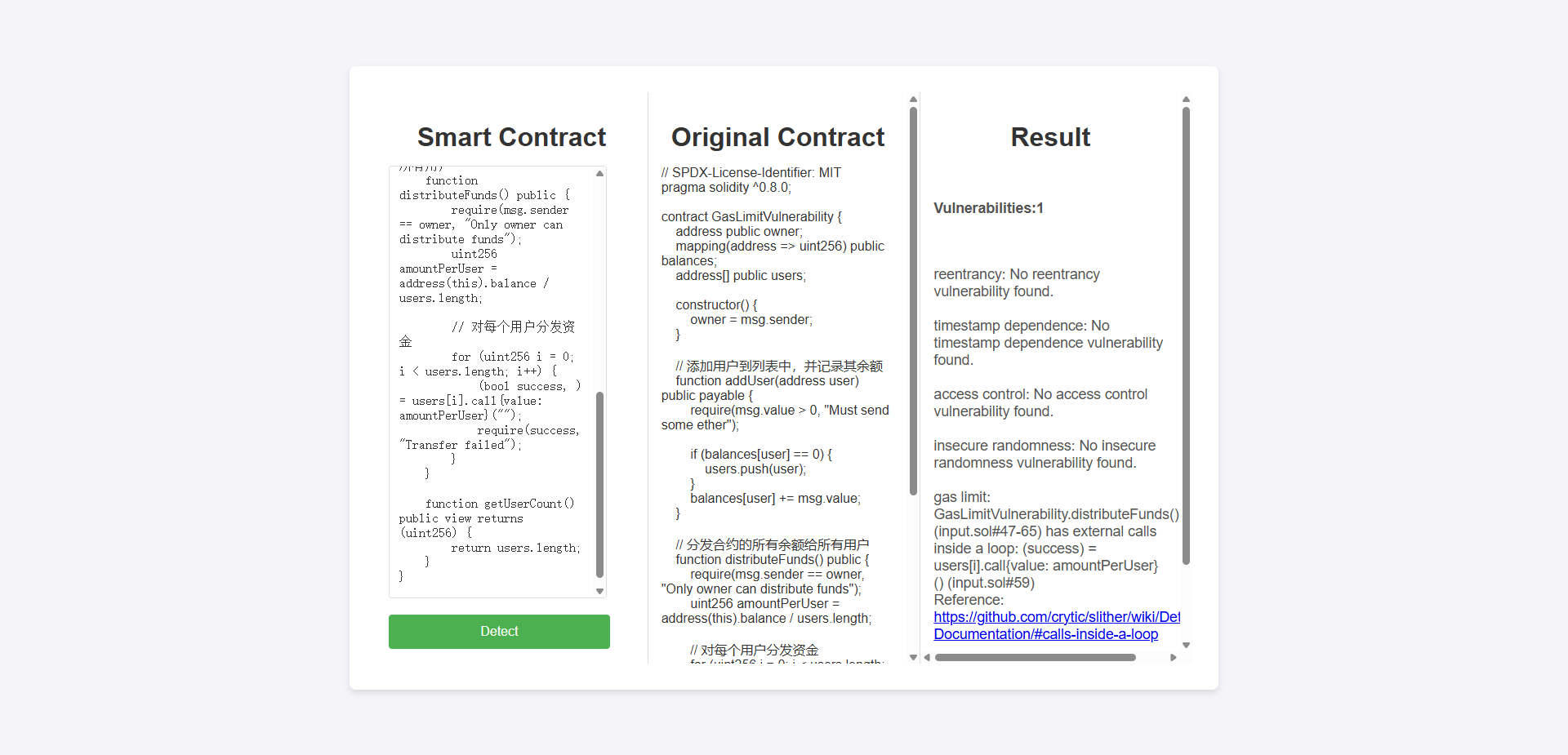
Picture 2: Vulnerability Detection Page

* 1. **Detect Results**
  2. **Slither Result for Access Control Vulnerability**



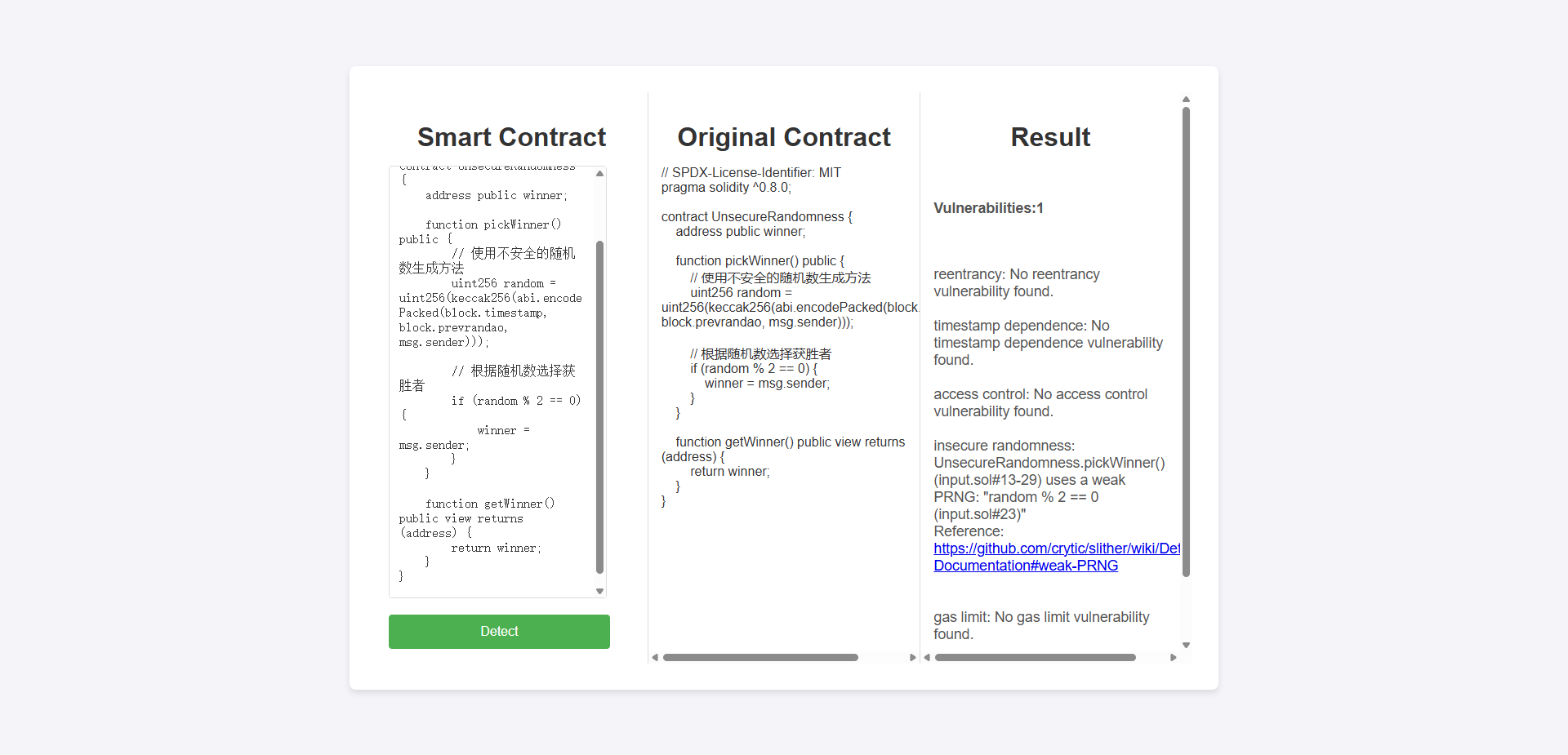
Picture 3: Detect access control

* 1. **Slither Result for Gas Limit Vulnerability**



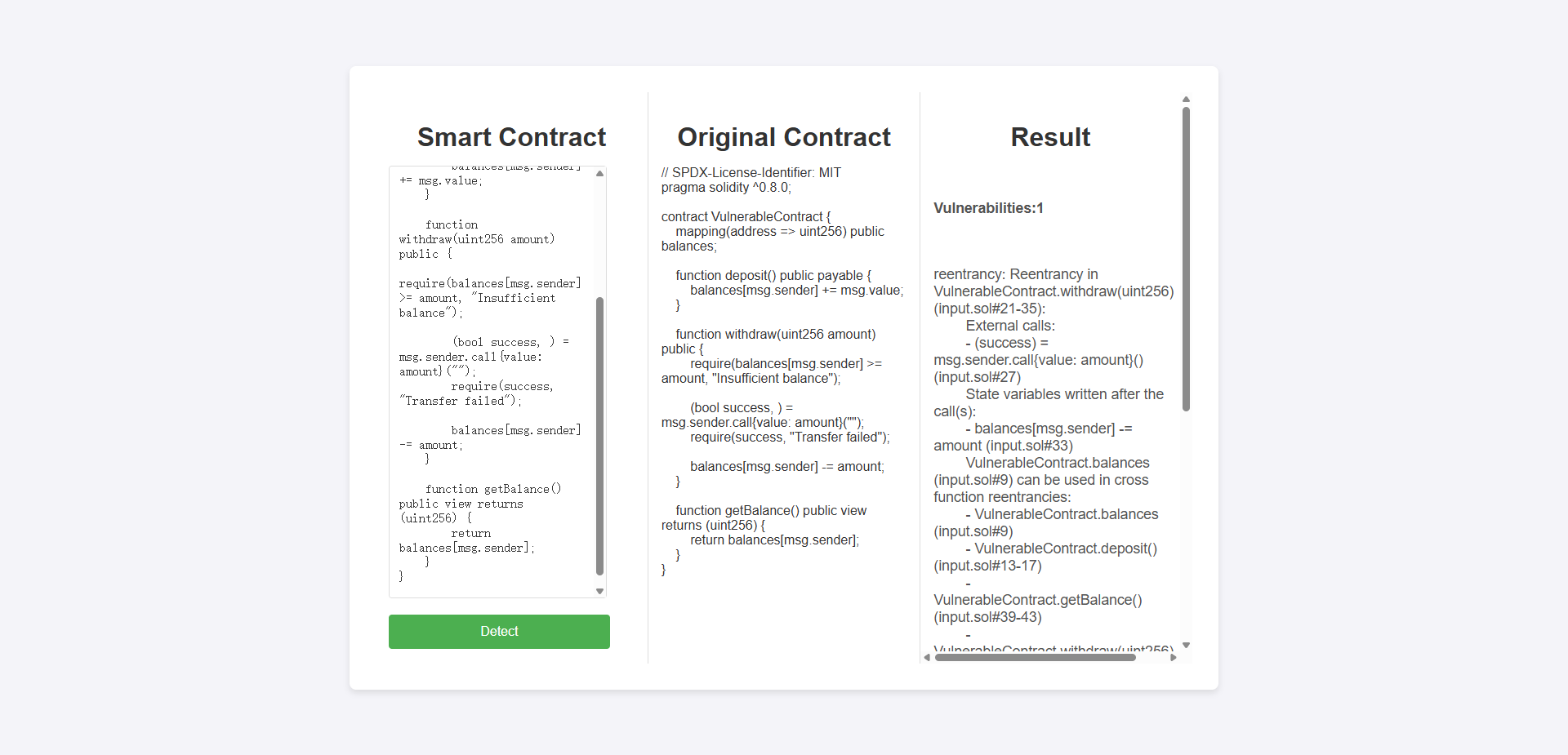
Picture 4: Detect gas limit

* 1. **Slither Result for Insecure Randomness Vulnerability**



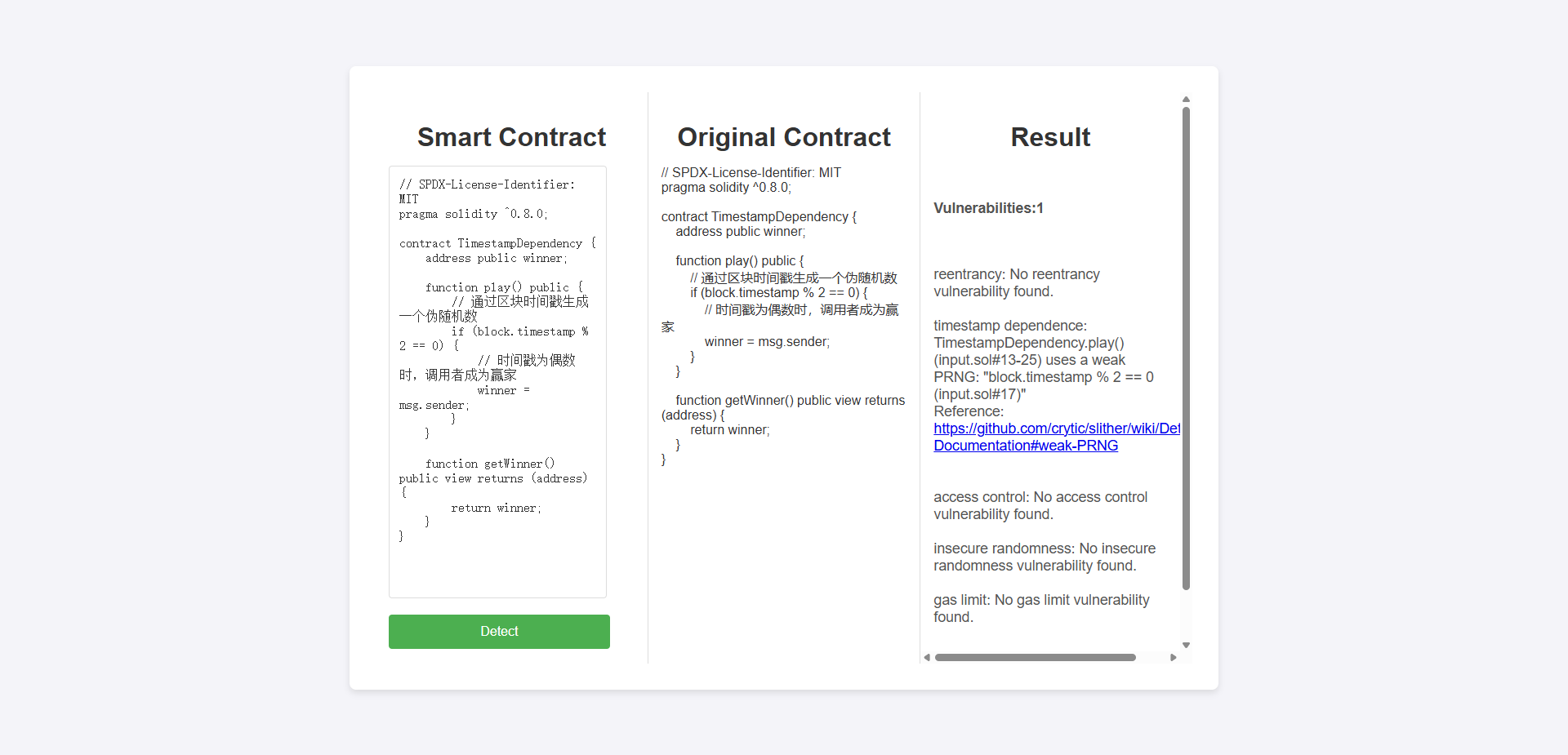
Picture 5: Detect insecure randomness

* 1. **Slither Result for Reentrancy Vulnerability**



Picture 6: Detect reentrancy

* 1. **Slither Result for Timestamp Dependence Vulnerability**



Picture 7: Detect timestamp dependence

1. **Conclusion**

In this project, we designed and implemented a vulnerability detection platform for Solidity smart contracts, based on the Flask web framework. By integrating the powerful Slither static analysis tool, our platform can automatically detect common security vulnerabilities in user-uploaded Solidity smart contracts. Through this platform, we aim to provide a convenient and efficient tool to identify potential security issues in smart contracts.

**Platform Objective** Our primary objective in developing this platform was to simplify the smart contract security audit process and offer an automated tool for detecting common vulnerabilities. Through deep integration with Slither, the platform is capable of quickly and accurately identifying typical security risks in smart contracts, helping developers uncover and fix these issues early to ensure the security of their contracts.

**Platform Advantages**

**Modular Design** Our system is designed with modularity in mind, with each vulnerability detection function encapsulated in independent modules. This not only makes the code easier to maintain but also facilitates future expansion. If additional vulnerability detection types are needed, we can easily implement and integrate new modules into the existing system.

**Vulnerability Reporting** The platform categorizes and counts detected vulnerabilities, displaying detailed results for each type. The results are dynamically rendered on the web interface, allowing users to quickly assess the security status of their smart contracts.

**Future** we plan to continue improving the platform’s functionality, expanding its detection capabilities, and optimizing the user experience. As smart contracts become more widely adopted in the blockchain space, we believe this platform will provide developers with a critical security safeguard, helping them build more robust and secure decentralized applications.

**Reference**

[1] Yueyue DAI, Ke ZHANG, Yan ZHANG. Blockchain empowered 6G[J]. Chinese Journal on Internet of Things, 2020, 4(1): 111-120.The Basic Concepts and Workings of Ethernet

<https://blog.csdn.net/weixin_39372311/article/details/141484391>

[2] What is Ethereum Virtual Machine (EVM) in Blockchain?

<https://medium.com/@orderlynetwork/what-is-ethereum-virtual-machine-evm-in-blockchain-e7db57167bea>

[3] What are Smart Contracts? Applications of Smart Contracts

<https://blog.csdn.net/tyxjolin/article/details/129823905>

[4] The Wonderful Use of Hash Algorithms in Ethernet

<https://learnblockchain.cn/article/867>

[5] What is Keccak256? Exploring the Cryptographic Hash Function and Its Use in Cryptocurrencies

<https://www.nervos.org/knowledge-base/what_is_keccak256_(explainCKBot)>

[6] Data Structures and Algorithms Used in Ethernet - Hash Functions - keccak256

<https://blog.csdn.net/qq_28053637/article/details/136571740>

[7] The current dilemma facing Ether (ETH)

<https://baijiahao.baidu.com/s?id=1808738261085987198>

[8] Ethernet's scalability crisis: exploring bottlenecks in the execution layer

<https://blog.csdn.net/feeltouch/article/details/140633326>

[9] What is Proof of Stake (PoS)

<https://m.163.com/dy/article/HARV5O9105535G1Y.html>

[10] Ethereum sharding technology

<https://zhuanlan.zhihu.com/p/37991389>

[11] Ethernet Layer 2 at a glance

<https://www.chinaz.com/2024/0520/1617722.shtml>

[12] OWASP Smart Contract Top 10

<https://owasp.org/www-project-smart-contract-top-10/>

[13] [Slither, the smart contract static analyzer](https://crytic.github.io/slither/slither.html)

https://github.com/crytic/slither

[14] [Ethereum Whitepaper](https://ethereum.org/en/whitepaper/#a-next-generation-smart-contract-and-decentralized-application-platform)

https://ethereum.org/en/whitepaper/#a-next-generation-smart-contract-and-decentralized-application-platform