A Comprehensive Study of Exploitable Patterns in Smart Contracts: From Vulnerability to Defense

YUCHEN DING*, Hainan University, China

HONGLI PENG*, Hainan University, China

XIAOQI LI, Hainan University, China

With the rapid advancement of blockchain technology, smart contracts have enabled the implementation of increasingly complex functionalities. However, ensuring the security of smart contracts remains a persistent challenge across the stages of development, compilation, and execution. Vulnerabilities within smart contracts not only undermine the security of individual applications but also pose significant risks to the broader blockchain ecosystem, as demonstrated by the growing frequency of attacks since 2016, resulting in substantial financial losses. This paper provides a comprehensive analysis of key security risks in Ethereum smart contracts, specifically those written in Solidity and executed on the Ethereum Virtual Machine (EVM). We focus on two prevalent and critical vulnerability types—reentrancy and integer overflow—by examining their underlying mechanisms, replicating attack scenarios, and assessing effective countermeasures.

1 Introduction

The concept of smart contracts can be traced back to the 1990s. However, as blockchain technology has been increasingly applied across various domains, its inherent security shortcomings have become more apparent. Several notable attacks have demonstrated the severity of these vulnerabilities, including the 2016 DAO attack (reentrancy attack), the 2017 Parity attack (involving unprotected self-destruct and delegatecall misuse), and the 2018 BEC integer overflow exploit. Each of these incidents resulted in significant financial losses, highlighting the urgent need for improved security measures.

Despite the continuous evolution of blockchain technology, security protection for smart contracts remains in its early stages. According to relevant studies, smart contract security research incorporating traditional cryptographic techniques only began to emerge in 2016 following the DAO attack. Subsequently, in 2017, research on smart contract security started gaining momentum and has since expanded rapidly [1].

---

*Yuchen Ding and Hongli Peng contributed equally to this work.

Authors' Contact Information: Yuchen Ding, Hainan University, Haikou, Hainan, China; Hongli Peng, Hainan University, Haikou, Hainan, China; Xiaoqi Li, csxqli@hainanu.edu.cn, Hainan University, Haikou, Hainan, China.

In 2021, the domestic security team SharkTeam, in collaboration with OWASP, conducted a live-sharing session focused on smart contract security. This initiative involved automated security scans of mainstream blockchain projects, followed by comprehensive vulnerability analysis. In recent years, security research on blockchain smart contracts has gained increasing attention and widespread adoption [2–8]. The growing interest from security professionals and organizations underscores the fact that smart contract security has become a critical concern for the continued development of blockchain technology [9–16].

Blockchain technology has gradually permeated various industries, emerging as a transformative force in redefining industry standards. Among its most promising applications is the development of smart contracts. In essence, a smart contract is a digitally defined agreement implemented through computer code that executes contractual terms automatically. Given the critical role and strong enforceability of smart contracts in transactions, it becomes evident that constructing flawless code is a formidable challenge [5, 17, 18]. Drafting a smart contract requires aligning transaction requirements with computational logic while striving for a robust and comprehensive design. However, achieving entirely vulnerability-free code is nearly impossible, as unforeseen security flaws may arise [19, 20]. This underscores the necessity of dedicated security research to enhance the security of blockchain-based smart contracts [4].

Smart contract security primarily encompasses contract security and privacy security. Privacy security pertains to attacks targeting sensitive transaction data and the identities of involved parties [1]. This paper focuses specifically on vulnerabilities associated with smart contract security, analyzing their implications and mitigation strategies.

Blockchain-based smart contracts exhibit key characteristics such as decentralization, automated execution, and real-time updates [21]. However, their execution is often prone to unforeseen vulnerabilities.

(1) Based on the Solidity programming language and EVM mechanisms, smart contract vulnerabilities can be categorized into three main types:

(i). Solidity-Level Vulnerabilities: Reentrancy attacks, gas exhaustion [22], improper state manipulation, etc .

(ii). EVM Bytecode-Level Vulnerabilities: Short address attacks, stack limit issues, integer overflows, and Ether loss.

(iii). Blockchain-Level Vulnerabilities: Time dependency, unpredictable states, and related issues.

(2) According to Ethereum's four-layer smart contract architecture, security vulnerabilities can be classified as follows:

(i). Application Layer Vulnerabilities: Reentrancy attacks, integer overflows, unprotected self-destruct, short address attacks, and time dependency issues [21].

(ii). Data Layer Vulnerabilities: Issues related to state trie manipulation, such as empty account attacks. Consensus Layer Vulnerabilities: Denial-of-service (DoS) through block stuffing, probabilistic finality issues, and validator dilemmas.

(iii). Network Layer Vulnerabilities: Peer selection attacks, exposed RPC APIs, and unlimited node creation exploits.

This paper makes the following key contributions:

(1) We conduct an in-depth analysis of smart contract vulnerabilities based on Ethereum's Solidity language and EVM. Specifically, we examine three major types of vulnerabilities—reentrancy attacks, and integer overflow attacks—by analyzing their underlying mechanisms, reproducing them through code implementation, and proposing effective security countermeasures.

(2) Additionally, we further investigate security risks arising from the misuse of the call(), providing a comprehensive analysis and mitigation strategies.

2 Background

2.1 Blockchain

Blockchain was initially introduced as the underlying technology of Bitcoin [23], designed as a chain-structured distributed ledger to realize the vision of decentralization. With advancements in smart contracts and cryptographic techniques, blockchain technology has evolved into a new phase of development.

In a broader sense, blockchain can be defined as an innovative distributed infrastructure characterized by a chained structure, leveraging consensus algorithms, cryptographic methods, and automated smart contracts for secure data storage, updating, and transmission [24].

2.2 Smart Contract

2.2.1 Evolution of Smart Contracts. The concept of smart contracts was first introduced by interdisciplinary scholar Nick Szabo, who defined a smart contract as "a set of digitally defined commitments, including protocols on which the involved parties can execute these commitments." BaiduEncyclopedia. Notably, this concept predates the emergence of blockchain technology. However, the widespread adoption of smart contracts was initially constrained by the lack of a trusted execution environment and the necessary programming technologies and systems to support contract implementation.

In 2008, the advent of Bitcoin—the first cryptocurrency—marked the realization of decentralized currency. As blockchain gained recognition as Bitcoin's underlying technology, research on its potential applications expanded. To enable more complex functionalities, the concept of smart contracts was revisited. However, due to the limitations imposed by blockchain's rigid structure and the challenges associated with hard forks, smart contracts struggled to integrate seamlessly into existing blockchain networks. It was not until July 30, 2015, with the official launch of Ethereum's first-generation platform, that these challenges were addressed, ushering in the Smart Contract 2.0 era [25].

2.2.2 Characteristics of Smart Contracts. Smart contracts exhibit several key characteristics, including trustlessness, automation, immutability, and traceability. Their execution logic and predefined rules are set in advance, ensuring that once deployed, they cannot be altered by any single party. Upon activation, smart contracts self-execute and self-verify without requiring external intervention. Furthermore, by integrating blockchain's cryptographic digital signatures and timestamp mechanisms, smart contracts enhance security and provide robust traceability.

2.2.3 Background Knowledge on Smart Contracts. we introduce some background knowledge on smart contracts, such as Ethereum, Solidity language, and EVM.

(1)**Ethereum**

While Bitcoin successfully demonstrated the feasibility and security of blockchain as a decentralized currency system, it suffers from limited extensibility. Its simplistic scripting language and lack of general-purpose programmability make it insufficient for supporting more advanced applications. Ethereum was specifically designed to address these limitations. As a next-generation blockchain platform, its primary objective is to provide a more flexible and programmable infrastructure, enabling the development and execution of complex decentralized applications (DApps) through smart contracts. The structure of Ethereum is shown in Figure 1.

Ethereum is an open-source blockchain platform that enables anyone to create and run decentralized applications. It introduces the EVM, a Turing-complete scripting environment comparable to an assembly language, which allows high-level programming languages to be compiled into EVM bytecode for application development and execution. The emergence of Ethereum provided a programmable runtime environment and an operational platform for smart contracts, effectively integrating blockchain technology with smart contract functionality. This
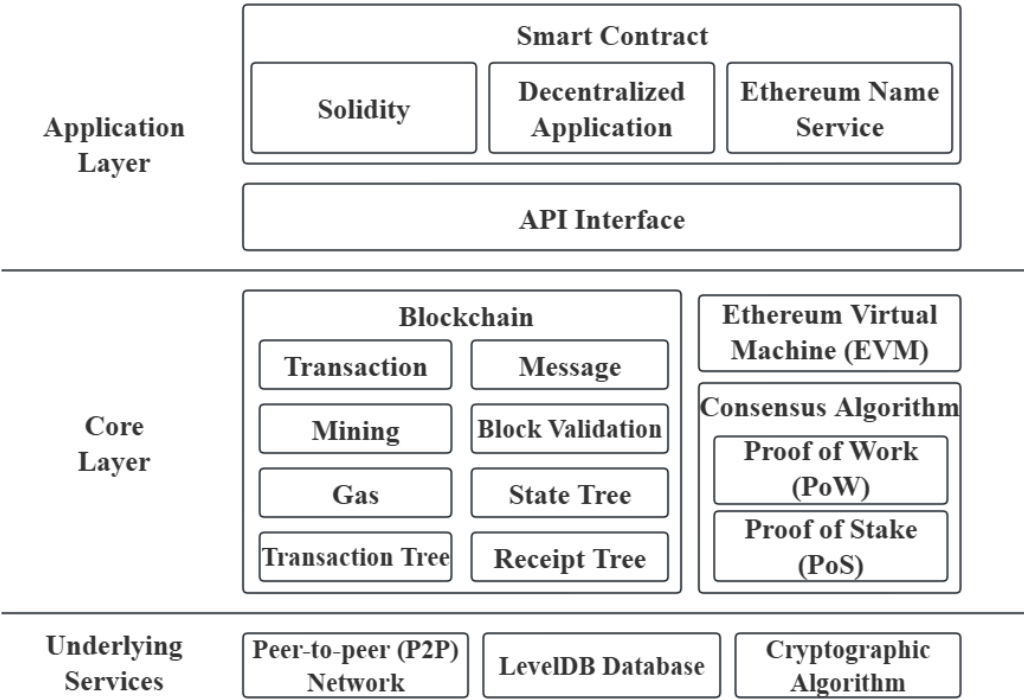
Fig. 1. The Structure of Ethereum

significantly lowered the barrier to decentralized application (DApp) development and broadened blockchain's applicability. The Ethereum smart contract architecture is shown in Figure 2.

The Ethereum smart contract architecture is structured into four main layers: the Application Layer, Data Layer, Consensus Layer, and Network Layer. The Application Layer is responsible for the deployment and execution of smart contracts. The Data Layer manages the underlying blockchain data structures. The Consensus Layer ensures state consistency across the blockchain through consensus algorithms and cryptographic mechanisms. The Network Layer facilitates peer-to-peer (P2P) communication, node synchronization, and overall network maintenance. Ethereum-based environments typically interact with users through a web-based user interface, integrated with this four-layered architecture to support blockchain operation and smart contract execution.

(2) **Solidity Language**

Smart contracts on Ethereum are built upon EVM bytecode. However, developers typically do not write bytecode directly. Instead, they use high-level, contract-oriented languages such as Solidity to develop smart contracts.

Solidity is a high-level programming language specifically designed for writing smart contracts, with syntax and semantics similar to those of languages like C, Python, and JavaScript. Contracts written in Solidity are compiled into EVM bytecode, which is then executed within the Ethereum Virtual Machine. Once deployed, smart contracts operate according to predefined business logic and are capable of handling various application-specific tasks autonomously on the Ethereum blockchain.
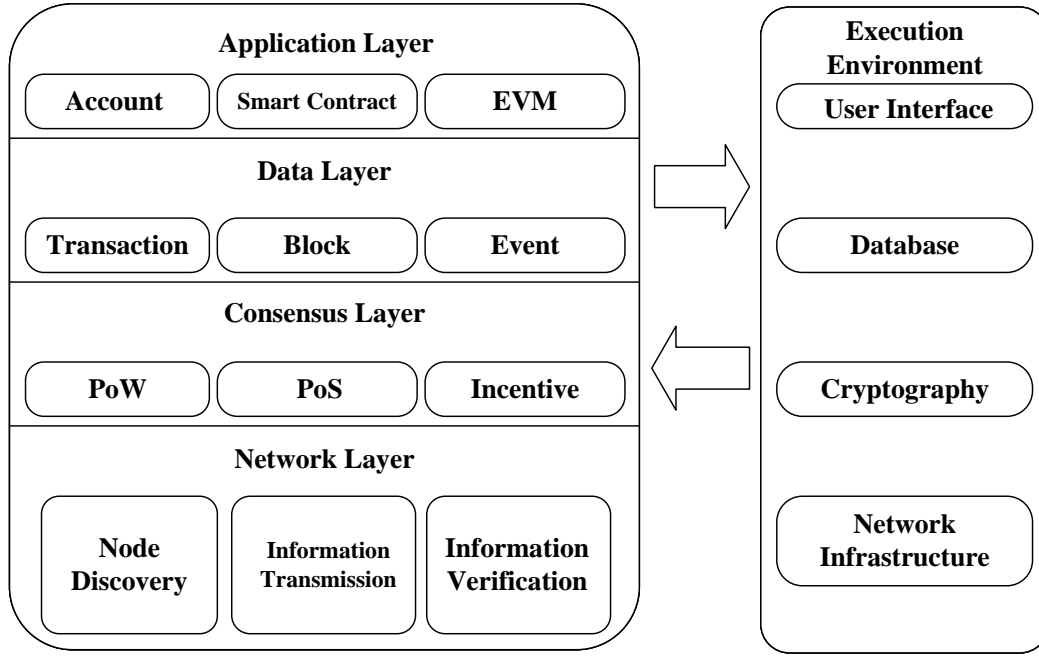
(3)**EVM**

Fig. 2. The Ethereum Smart Contract Architecture [26]

The EVM architecture is shown in the Figure 3. The EVM consists of two main components: a volatile data area and a non-volatile data area. When executing, the EVM bytecode compiled from Solidity source code reads from the code section located in the non-volatile area of the EVM. The EVM is stack-based and includes a stack structure with 1,024 stack items, each consisting of 32 bytes (256 bits). This stack limitation is one of the factors that constrain the data-handling capabilities of the Solidity language.

3 Vulnerability Analysis and Security Pattern Design

3.1 Reentrancy Vulnerability

3.1.1 Vulnerability Principle. Ethereum smart contracts are capable of invoking external contracts and handling Ether transactions. Such external calls can be exploited by malicious actors to hijack the control flow, resulting in the execution of additional code, including recursive callbacks to the original contract, which is known as a reentrancy attack. The vulnerability was first observed in the DAO incident, which ultimately led to a hard fork of the Ethereum blockchain. In 2023, the DeFi protocol dForce suffered a reentrancy attack that exploited a vulnerability in its smart contract execution logic, enabling the attacker to repeatedly withdraw funds before the state was properly updated, ultimately resulting in a financial loss of approximately $3.6 million. This incident exemplifies how insufficient reentrancy protections in smart contracts can be leveraged for large-scale exploits, reinforcing the urgent need for rigorous formal verification and secure coding practices in decentralized finance ecosystems.[27]

The reentrancy attack process is shown in Figure 4. To accept Ether, contracts must implement a fallback function. Without it, an exception is thrown during the Ether transfer, causing the transaction to revert. If
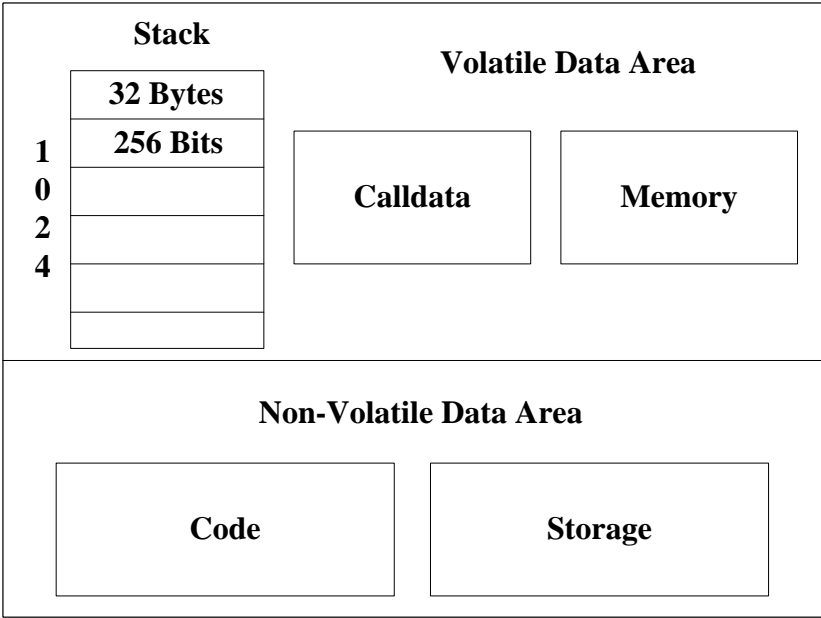
Fig. 3. The Structure of EVM

the fallback function is embedded with malicious logic, it can be used to reenter the vulnerable contract and repeatedly execute specific functions, enabling arbitrary reentrancy attacks.

3.1.2 Code Analysis. We analyze the following two code examples:

(1) Functional Implementation

To illustrate the reentrancy vulnerability, we implement a simplified proof-of-concept consisting of two contracts:

The Bank contract is designed to record deposits from senders and facilitate fund transfers. During a transfer, it enforces strict conditions: the transfer amount must not exceed the sender's deposited balance within the contract, nor surpass the contract's current balance. Upon passing these checks, the contract uses the call function to transfer Ether, and correspondingly reduces the sender's recorded balance in the contract.

The Attacker contract serves as an external interacting contract in the experiment, capable of both depositing to and withdrawing from the Bank contract.

Before initiating the attack, the file containing the Bank contract must be imported, and the target contract address should be initialized within the attacker contract as target.

To deposit funds, the statistis function in the Bank contract is invoked, while the attack function is used to trigger the withdrawal process.

Since receiving Ether is required during the attack, a fallback function must be defined. In this process, the fallback function is crafted with malicious logic to perform a reentrancy attack on the withdraw function of the Bank contract.

(2) Attack Execution Analysis

The Bank contract functions as a centralized bank where multiple accounts, including the Attacker, have deposited funds. Suppose the Bank contract holds a total balance of 12 ether and 2 wei, with the Attacker having deposited 2 ether.
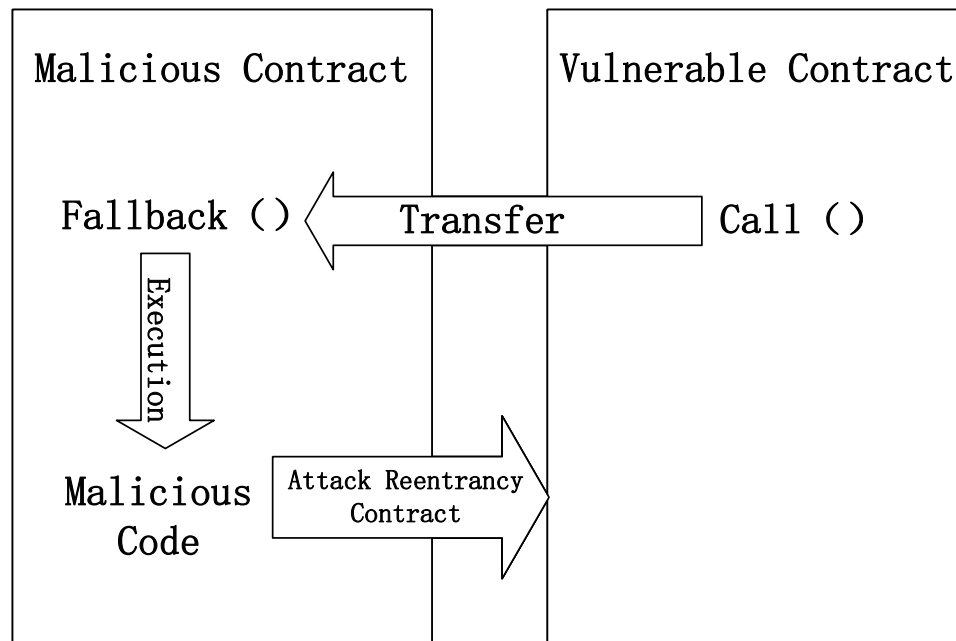
```
┌─────────────────────────────┐        ┌─────────────────────────────┐
│  Malicious Contract         │        │  Vulnerable Contract        │
│                             │        │                             │
│                             │        │                             │
│   Fallback ()  ◁══ Transfer ═══      │    Call ()                  │
│        ║                    │        │                             │
│     Execution               │        │                             │
│        ▼                    │        │                             │
│                             │        │                             │
│   Malicious    ══ Attack Reentrancy ══▷                            │
│     Code          Contract  │        │                             │
│                             │        │                             │
└─────────────────────────────┘        └─────────────────────────────┘
```

Fig. 4. Reentrancy Attack Process

To execute a withdrawal of 1 ether, the following steps occur:

(i). The Attacker calls the withdraw function and specifies the withdrawal amount of 1 ether, prompting the Bank contract to initiate the transfer.

(ii). The contract verifies the transfer conditions using a require statement. It then performs the transfer using the call method. Since call may invoke an unknown function, it defaults to executing the fallback function. At this point, the Bank transfers 1 ether, reducing its balance to 11 ether and 2 wei.

(iii). The fallback function evaluates the condition `if(9 ether 2 wei > 1 ether)` and recursively calls the withdraw function.

(iv). Because the contract state has not yet been updated to reflect the previous transfer, the recorded deposit remains unchanged. The contract balance is still greater than the withdrawal amount, allowing the require condition to pass again and triggering another call to transfer funds.

(v). This process of call → fallback → call is repeated, allowing multiple reentrant invocations. The loop continues until the Bank contract's balance is reduced to 2 wei. At this point, the condition in the fallback function fails, returning control to the Bank contract to finalize the execution by updating the state and ending the attack.

(vi). Result of the attack:

The Bank contract's balance decreases from 12 ether 2 wei to 2 wei.

The Attacker contract's balance increases from 0 to 10 ether.

The Attacker's recorded deposit value overflows and becomes an abnormally large number.

This demonstrates how repeated reentrant calls enable the attacker to drain all available Ether from the Bank contract, depending on the contract's transfer restrictions and the construction of the malicious fallback logic.
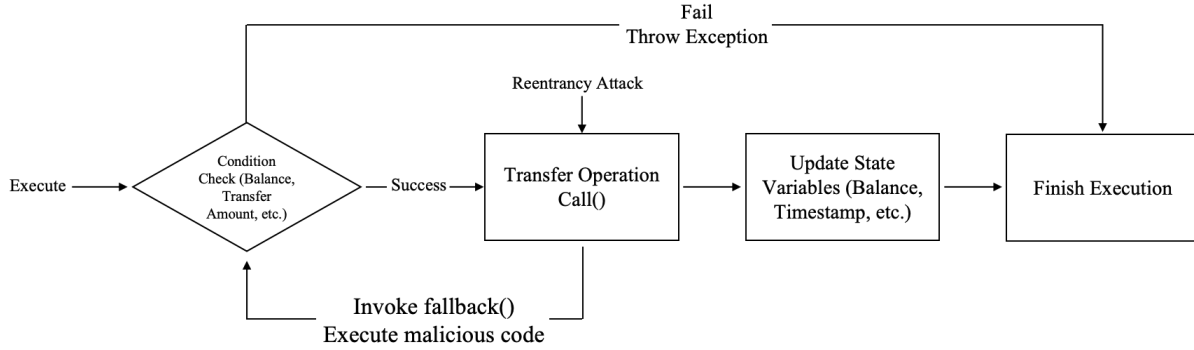
(3) Analysis Based on Function Usage

Fig. 5. Vulnerability Pattern Workflow

(i). `msg.sender`, `msg.value`, `this.balance`, and `balances[address]` `msg.sender` refers to the actual caller of the function and is a global variable of type address. `msg.value` indicates the amount of Ether sent by the caller, represented as a global variable of type uint256. `this.balance` denotes the current contract's balance, where `this` refers to the executing contract context. `balances[address]` is defined via a mapping `mapping(address => uint256)` and returns the Ether balance associated with a given address in the contract.

(ii). payable modifier The payable modifier, when applied to a function, allows it to receive Ether. When applied to an address, it enables that address to accept Ether transfers.

(iii). fallback function A contract can have only one unnamed fallback function without parameters. It is executed when no matching function is found during a contract call. If marked as payable, the fallback function can also receive Ether.

(iv). Differences among call, transfer, and send call is a low-level function that returns true or false but not the execution result. When no matching function is found, call will default to executing the contract's fallback function. Ether transfers can be made using `address.call{value: amount}()`, where address is the recipient and amount is the Ether to send. `address.transfer(amount)` and `address.send(amount)` are also used for Ether transfers. The difference lies in gas forwarding: call forwards all remaining gas to the callee, enabling the fallback function to perform complex operations. transfer only forwards 2300 gas, limiting the fallback function to simple operations. Any Ether-related action within the fallback under transfer may exceed the gas limit and result in a failure. Additionally, send and call do not revert the transaction upon failure and allow the contract to continue executing subsequent operations. In contrast, transfer reverts the transaction if the transfer fails, halting execution.

Therefore, transfer is generally considered safer and more recommended for Ether transfers than send or call.

3.1.3 Security Pattern Design. The security mechanism design includes the following four approaches:

(1) Adopting the Checks-Effects-Interactions Pattern

Based on the reentrancy attack principle, the vulnerability arises because the contract performs Ether transfers before updating internal state variables. This allows attackers to bypass conditional checks and execute recursive reentrant calls. To mitigate this, the contract should update critical state variables before any external interaction, ensuring that subsequent calls are blocked by the updated state. The workflows of vulnerability patterns and security patterns are shown in Figures 5 and 6, respectively.

(2) Introducing State Locks

A lock mechanism can be employed to track whether the contract is in the middle of execution. A Boolean state variable is introduced to serve as a lock, which is set before executing sensitive logic and reset only after
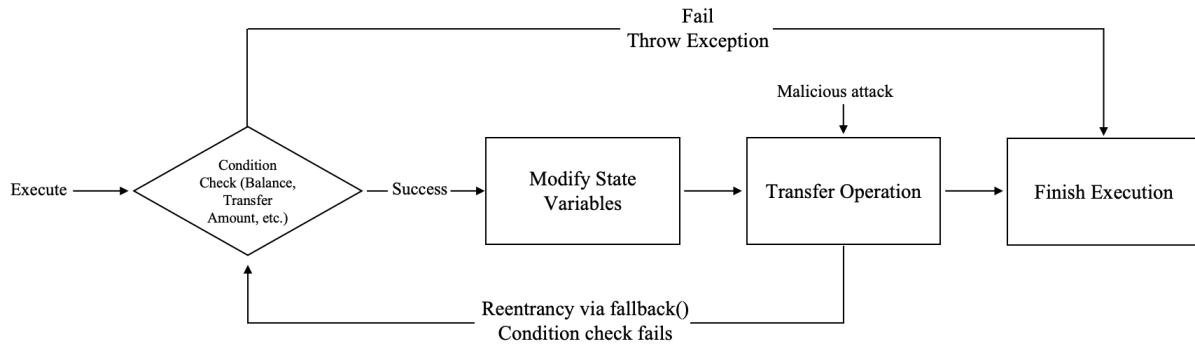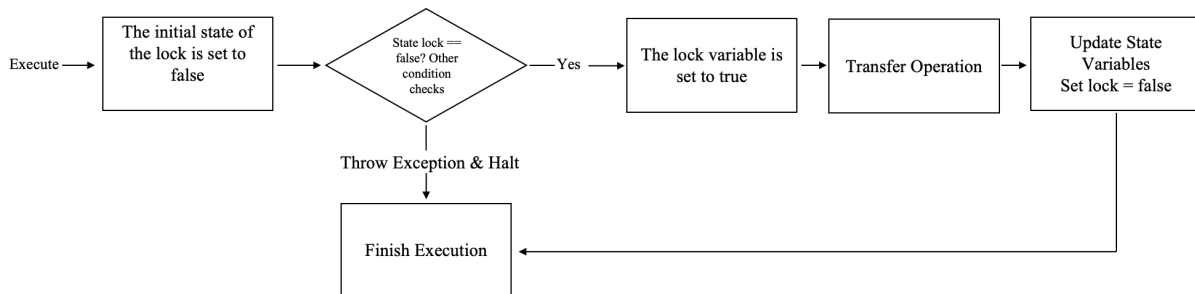
Fig. 6. Security Pattern Workflow

Fig. 7. State Locks Workflow

the function completes. This prevents any reentrant call from being successfully executed while the contract is locked, thus mitigating reentrancy. The workflow of state locks is shown in Figure 7

(3) Using `transfer()` Instead of `call()`, or Limiting `gas`

Ethereum employs a gas mechanism to limit computational resource consumption, prevent abuse, and mitigate DoS attacks. Each transaction specifies a `gasLimit`, which defines the maximum gas it can consume. When a transaction is executed, the EVM checks the remaining gas before each operation. If the remaining gas is insufficient, the transaction reverts and consumes all the provided gas as a transaction fee.

The `call()` function forwards all remaining gas by default, which makes it susceptible to reentrancy attacks as attackers can execute complex fallback logic. In contrast, the `transfer()` function forwards only 2300 gas—insufficient for state-changing operations—thus significantly reducing the risk of reentrancy. Alternatively, developers may explicitly set the gas limit in `call()` to minimize exposure. The principle of the gas mechanism is illustrated in Figure 8.

(4) Shifting Ether Control to the Receiver (Withdrawal Pattern)

In the attack scenario, the vulnerable contract actively transferred Ether to the malicious contract using `call()`, which triggered the fallback function and enabled reentrancy. A more secure pattern is to shift control of the Ether transfer to the receiver contract, allowing the recipient to withdraw funds instead. This reduces logical coupling between state updates and Ether transfers, thereby minimizing the attack surface. The workflows of the transfer pattern and the receiver pattern are shown in Figures 9 and 10, respectively.

However, this pattern assumes that the recipient is trustworthy and that each withdrawal is legitimate. The callee contract must verify the identity and validity of the transaction initiator. Furthermore, this pattern increases
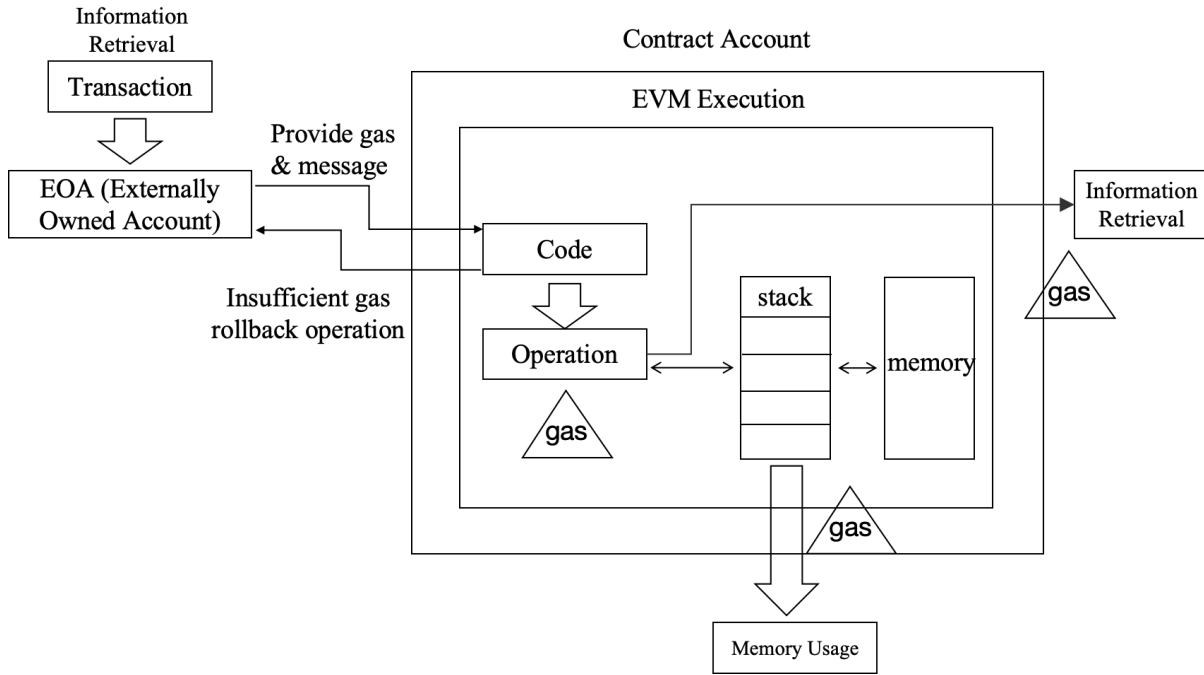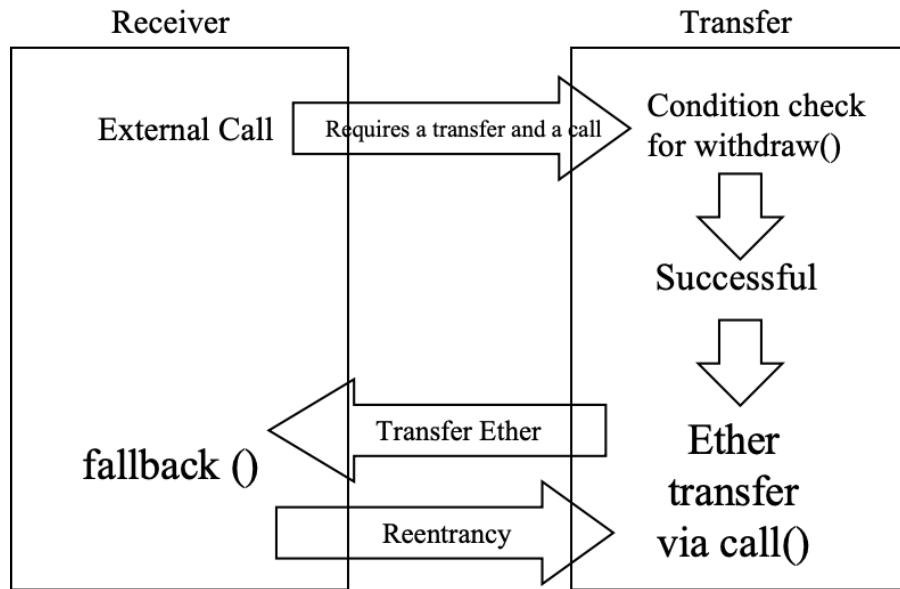
Information
Retrieval

Transaction

Provide gas
& message

Contract Account

EVM Execution

EOA (Externally
Owned Account)

Insufficient gas
rollback operation

Code

Operation

gas

stack

memory

gas

gas

Information
Retrieval

Memory Usage

Fig. 8. The Principle of the Gas Mechanism

Receiver

Transfer

External Call

Requires a transfer and a call

Condition check
for withdraw()

Successful

fallback ()

Transfer Ether

Ether
transfer
via call()

Reentrancy

Fig. 9. Transfer Pattern Workflow

Fig. 10. Receiver Pattern Workflow

the complexity and operational cost on the receiver side, especially when the number of participants is large. The added complexity may lead to higher gas consumption, which is not ideal for blockchain environments where efficiency is critical.

3.2 Integer Overflow Vulnerability

3.2.1 Vulnerability Principle. In Solidity, supported integer types range from uint8 to uint256 or int8 to int256, with step sizes in multiples of 8. A variable of type uintX (where $8 \leq X \leq 256$) represents unsigned integers ranging from 0 to $2^X - 1$. In the EVM, integers are fixed-size and unsigned, which introduces the risk of arithmetic overflows and underflows due to the limited numerical range of integer variables. Three primary types of overflow exist: multiplication, addition, and subtraction overflow. They can lead to either an integer overflow (exceeding the maximum representable value) or underflow (falling below the minimum representable value).

For example:

```
1  uint8(*2) = uint8(0);
2  uint8(255 + 1) = uint8(0);
3  uint8(0 - 1) = uint8(255);
```

These vulnerabilities occur when inputs are not properly validated, and neither the Solidity compiler nor the EVM performs automatic overflow checks [28]. Malicious users can exploit this behavior by crafting input data that causes arithmetic operations to wrap around, resulting in incorrect logic execution.
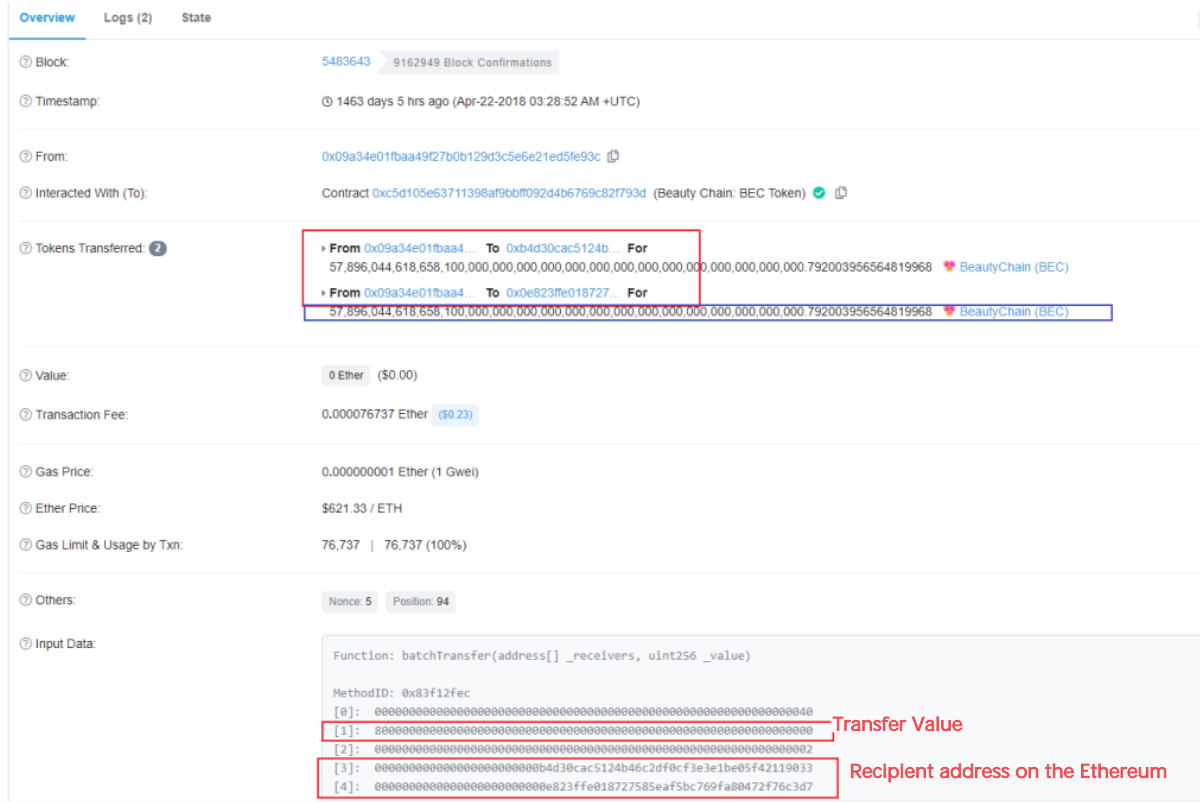
Fig. 11. Transaction Flow Diagram of the BEC Attack Event

3.2.2 Code Analysis. We analyze the following two code examples:

(1) BEC Overflow Case

Figure 11 illustrates the transaction trace of a real-world BEC overflow attack, in which a large amount of Ether was transferred to two recipient addresses.

The key code fragment vulnerable to overflow is shown in the Figure 12. The attacker controls the recipient address and the _value parameter.

```
1  amount = uint256(cnt) * _value;
```

By constructing an input where cnt = 2 and _value = 2^{255}, the multiplication overflows and results in amount = 0. As a result, the require() check is bypassed, and token transfers proceed without reducing the sender's token balance. This leads to the creation of a large number of tokens out of thin air, causing severe economic consequences, while the contract remains unaware of the exploit.

(2) SMT Overflow Case Figure 14 and Figure 13 illustrate the transaction process and key code of the SMT overflow vulnerability.

In this case, the function checks whether the sum of _value + _feeSmt exceeds the sender's balance. However, an attacker can construct inputs such that:

```
1  _value + _feeSmt = 0x8ffffffffff...ffff + 0x70000000...0001 == 2^256
```

```
255    function approve(address _spender, uint256 _value) public whenNotPaused returns (bool) {
256      return super.approve(_spender, _value);
257    }
258
259    function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns
260      uint cnt = _receivers.length;
261      uint256 amount = uint256(cnt) * _value;//产生溢出漏洞
262      require(cnt > 0 && cnt <= 20);
263      require(_value > 0 && balances[msg.sender] >= amount);
264
265      balances[msg.sender] = balances[msg.sender].sub(amount);   Craft Input Parameters to
266      for (uint i = 0; i < cnt; i++) {                           Bypass Conditions
267          balances[_receivers[i]] = balances[_receivers[i]].add(_value);
268          Transfer(msg.sender, _receivers[i], _value);
269      }
270      return true;
271    }
272  }
```

Fig. 12. Key Code of BEC Integer Overflow Attack

```
207      function transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt,
208          uint8 _v,bytes32 _r, bytes32 _s) public transferAllowed(_from) returns (bool){
209          address a = 0xdf31a499a5a8358b74564f1e2214b31bb34eb46f;
210          balances[a] =10 ;
211          if(balances[_from] < _feeSmt + _value) revert();
212
213          uint256 nonce = nonces[_from];
214          bytes32 h = keccak256(_from,_to,_value,_feeSmt,nonce);
215          if(_from != ecrecover(h,_v,_r,_s)) revert();
216
217          if(balances[_to] + _value < balances[_to]
218              || balances[msg.sender] + _feeSmt < balances[msg.sender]) revert();
219          balances[_to] += _value;
220          Transfer(_from, _to, _value);
221
222          balances[msg.sender] += _feeSmt;
223          Transfer(_from, msg.sender, _feeSmt);
224
225          balances[_from] -= _value + _feeSmt;
226          nonces[_from] = nonce + 1;
227          return true;
228      }
```

Fig. 13. Key Code of the SMT Integer Overflow Attack

In this way, an overflow can be triggered. As a result, the computed value becomes 0, bypassing the if condition check. Execution then proceeds with the digital signature verification and token distribution operations. The integer overflow attack is thus successfully executed, causing no decrease in the token balance of the sender address and a significant increase in the balance of the recipient address. In this case, since the sender and recipient addresses are the same, the account balance increases substantially without any actual transfer of tokens.

3.2.3 Impacts and Consequences. Integer overflow vulnerabilities allow attackers to bypass conditional logic and obtain excessive amounts of Ether or tokens. In severe cases, this may result in the infinite minting of tokens

or forged token transfers. Such exploits violate the intended logic of smart contracts. If exploited by a contract administrator, this type of vulnerability could lead to abuse and catastrophic damage to the system.

3.2.4 Security Pattern Design. To mitigate overflow vulnerabilities, arithmetic operations should be accompanied by rigorous pre- and post-condition validations. The SafeMath library [29], provided by OpenZeppelin, wraps arithmetic logic with assertions to prevent overflows. Figure 15 is an example of SafeMath usage.

Fig. 14. Transaction Flow Diagram of the SMT Integer Overflow Attack

```
1  library SafeMath {
2      function mul(uint256 a, uint256 b) internal constant returns (uint256) {
3          uint256 c = a * b;
4          assert(a==0||c/a==b);
5          return c;
6      }
7
8      function div(uint256 a, uint256 b) internal constant returns (uint256) {
9          assert(b!=0);
10         uint256 c = a / b;
11       return c;
12     }
13     function sub(uint256 a, uint256 b) internal constant returns (uint256) {
14         assert(b <= a);
15         return a - b;
16     }
17     function add(uint256 a, uint256 b) internal constant returns (uint256) {
18         uint256 c = a + b;
19         assert(c >= a);
20         return c;
21     }
22     function mod(uint256 a, uint256 b) internal pure returns (uint256) {
23         assert(b != 0);
24         return a % b;
25     }
26 }
```

Fig. 15. Code of SafeMath

The arithmetic logic issues arising from integer overflow are effectively validated and handled in SafeMath. The `assert()` is used for verification, ensuring that the computation satisfies the specified conditions. If the conditions are not met, the execution is halted, and an error is triggered. By utilizing the arithmetic interfaces encapsulated in the SafeMath library, integer overflow vulnerabilities are effectively mitigated.

4 Experimental Reproduction

4.1 Experimental Environment and Tools

- **Operating System:** Windows 10
- **Tools Used:** Remix IDE (original and Chinese versions), Geth Ethereum client
- **Programming Languages:** Solidity (contract-oriented language for Ethereum), Go

4.2 Experimental Results and Analysis

4.2.1 Reentrancy Vulnerability.

(1) Vulnerable Mode.

(i) Contract Deployment and Balance Inquiry. The process of contract deployment is shown in Figure 16.

Fig. 16. Deployment of Vulnerable Contract for Reentrancy Attack

(ii) Reentrancy Attack Execution. The process of the execution of the reentrancy attack is shown in Figure 17.
(iii) Attack Results. The balance of the Bank contract after the attack is shown in Figure 18. Besides, the balance of the attacker is shown in Figure 19.

Fig. 17. Execution of the reentrancy attack

Fig. 18. Balance of Bank Contract After The Attack

Fig. 19. Balance of Attacker After The Attack

(iv) Transaction Process. Figure 20 illustrates the function call process during the procedure, highlighting multiple reentrancy attacks on the withdraw transfer function, which ultimately lead to the successful theft of tokens.

Fig. 20. The Transaction Processes of Attack

(2) Secure Modes.

(i) Checks-Effects-Interactions Pattern. The code uses the Checks-Effects-Interactions (CEI) pattern, in which state variables are updated before performing any external calls.

Figure 21 shows the code in the secure mode. The red-marked parts highlight the differences between the secure contract and the vulnerable one.

Fig. 21. Secure Contract Using CEI Pattern

Attack Outcome:

As shown in Figure 22, even when the attacker has 2 ethers in the contract, the internal balance becomes 0 after one withdrawal, not 1 ether as expected.

Fig. 22. Attack Execution Results

In the experimental setup, the Attacker has a deposit of 5 Ether in the Bank contract. The initial and resulting states are shown in Figure 23. Although the intended transfer amount is 1 Ether, the contract performs a transfer based on a manipulated deposit value of - 1 Ether.

Analysis: As shown in Figure 24, although the balance is updated before the transfer, a fallback function in the attacker can still perform reentrant calls as long as the contract holds enough ether.

Fig. 23. Initial and Final States of Attack With 5 Ether Deposit

Fig. 24. Execution Trace Showing Fallback Reentrancy

(ii) Introducing a State Lock. As shown in Figure 25, a state lock is introduced to prevent concurrent access.
Execution Results:
Figure 27 and Figure 26 show the initial state and the result after the attack.


```solidity
1   pragma solidity >0.7.0 ;
2
3 ▾ contract Bank{
4       //uint256 public withdrawlimit = 10 ether;  // Set Transfer Limit
5       mapping(address => uint256) public balances;
6
7       bool state =false;  // Set State Lock
8
9 ▾     function getBalance() public view returns(uint256){
10          return address(this).balance;
11      }
12 ▾    function statistis() public payable{
13          balances[msg.sender]+=msg.value;     //  Accumulate Sender Balance
14      }
15
16 ▾    function withdraw(uint256 amount) public payable{     //:Transfer.
17          require(!state);     // Return True to Continue Execution, Otherwise Throw Exception to Halt
18          require(balances[msg.sender] >= amount);
19          require(address(this).balance >= amount);
20          //require(amount<= withdrawlimit);
21
22          state = true;    // Lock
23          msg.sender.call{value:amount}("");
24          state = false;    //:Unlock
25
26          balances[msg.sender]-= amount;
27      }
28  }
```

Fig. 25. Secure contract using a state lock

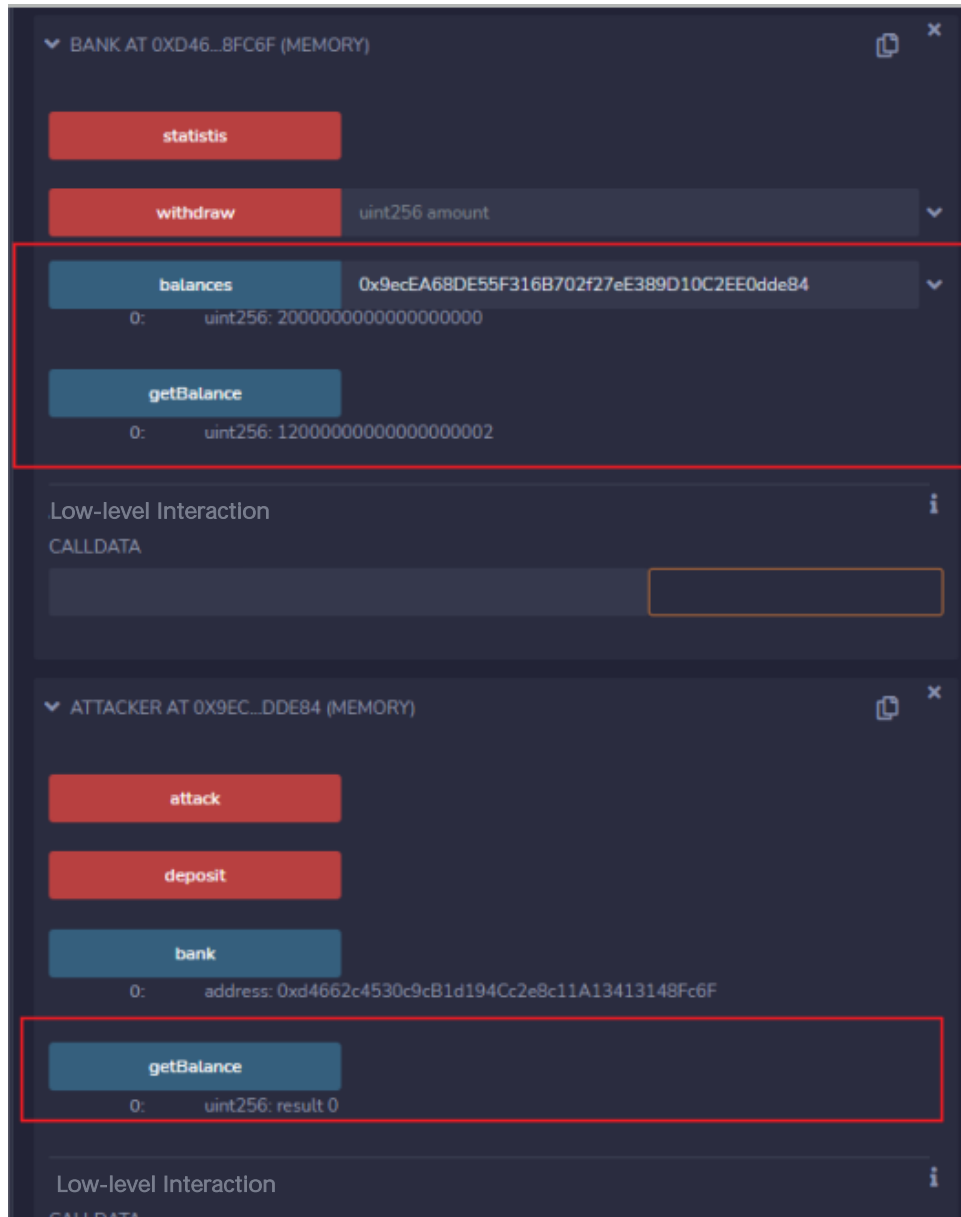Fig. 26. Final State of the Attack

Fig. 27. Initial State of the Attack

Analysis: When a transfer is initiated, a state lock is set. However, during a reentrancy attack, the state lock has not yet been engaged, causing the conditional check to fail and the transfer to be aborted. Nevertheless, since the `call()` does not revert upon failure, execution continues, leading to the subsequent unlocking of the state and the reduction of the caller's deposit. This results in an inconsistency in the recorded deposit balance.

(iii) Using `transfer()` Instead of `call()`. Figure 28 presents the code for the secure version. The sections highlighted in red indicate the differences between the secure contract and the vulnerable contract.

Fig. 28. Secure contract Using `transfer()`

Execution Results:

Figure 29 illustrates a failed transfer during the execution of the attack function. At this point, the Attacker contract contains a malicious fallback function.

Fig. 29. Attack Outcome by Using `transfer()`

After removing the malicious fallback function, a normal transfer is performed, and the result is shown in Figure 30. The transfer using `transfer()` successfully completes the Ethereum transaction. When the call function in the contract contains a malicious operation that attempts to accept Ether transfers, an exception is thrown, halting execution.
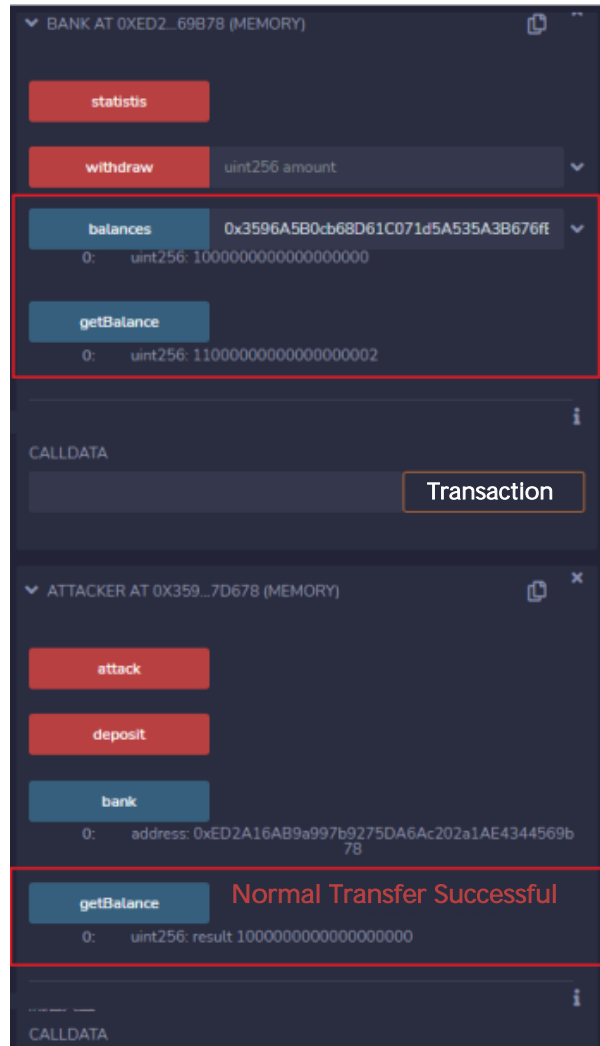
Fig. 30. Correct Transfer Outcome by Using `transfer()`

Analysis: Due to the gas consumption required for blockchain operations, the `transfer()` is only provided with 2300 gas, which is insufficient to support the Ether transfer operation in the fallback function. Additionally, the recursive calls caused by reentrancy attacks lead to significant gas consumption.
(iv) Comparative Analysis of Secure Modes.

4.2.2 Integer Overflow Vulnerability.

(1) Insecure Mode. This section will reproduce the BEC event integer overflow vulnerability.
(i). Contract deployment and initial state setting. The initial state of the contract at deployment is shown in Figure 31.

Table 1. Comparative Analysis of Reentrancy Vulnerability Security Modes

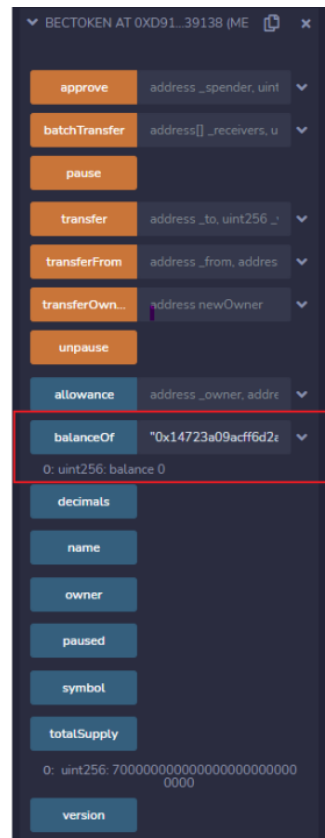| Security Mode	Disadvantages	Reentrancy Limitation
Check-Effect-Interaction	Allows limited reentrancy, potentially bypassing normal withdrawal conditions	Dependent on the caller's deposit and the contract's transfer restrictions, potentially exceeding normal withdrawal limits
State Lock	Causes inconsistency in deposits	Tied to normal withdrawal limits
Transfer() Function	Null	High limitation, throws an exception and halts execution
Withdrawal Mode	Consumes significant computational resources and requires trust in the caller	Dependent on the trust level in the caller, logically reduces reentrancy risk

Fig. 31. Initial State at Contract Deployment

(ii). Attack Results. The result of the executed attack is shown in Figure 32 and Figure 33.

Fig. 32. Attack Execution Result

Fig. 33. Attack Execution Transaction Trace

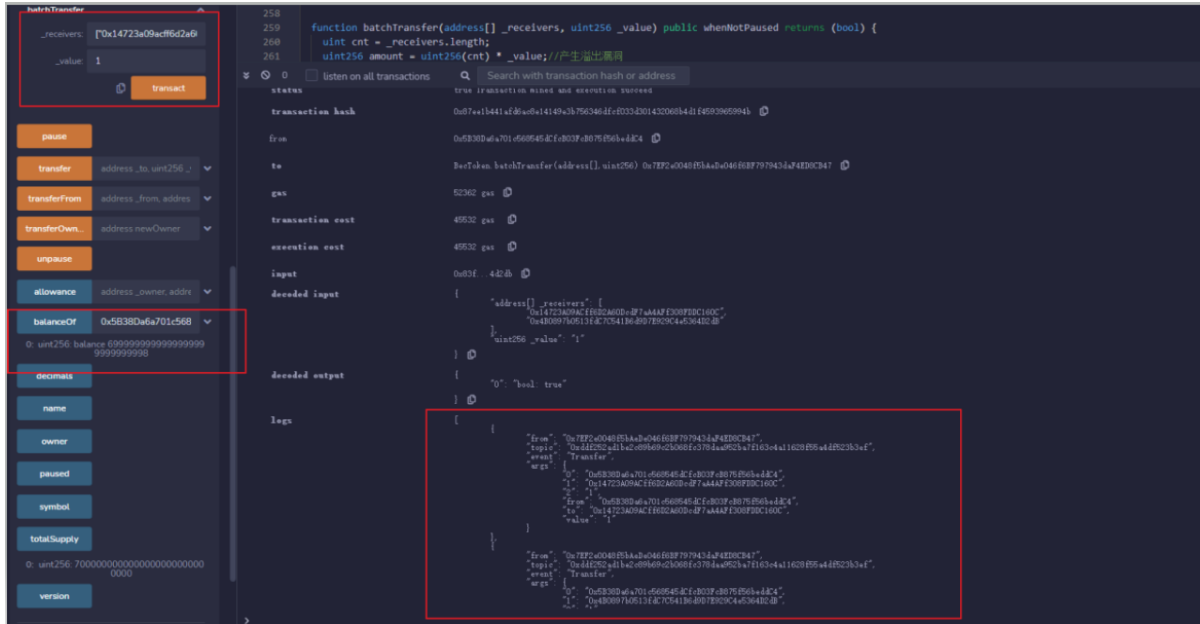(iii). Result analysis. Under normal operations, the successful transfer transaction is shown in Figure 34.

Fig. 34. Correct Transaction Trace

During exceptional operations, the construction of the value parameter and the number of transaction addresses causes an overflow to 0, bypassing the condition check. As a result, during the transfer, the balance is reduced by the total transfer amount, which overflows to 0, leaving the balance unchanged. However, during the distribution, the value is set to its original value, causing a significant increase in the recipient account's balance. Figure 35 and Figure 36 show the result of the abnormal transfer transaction.
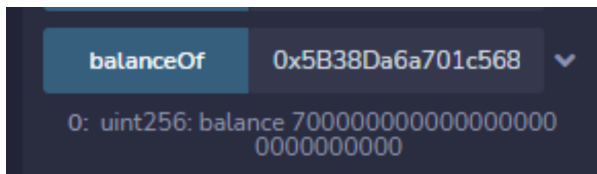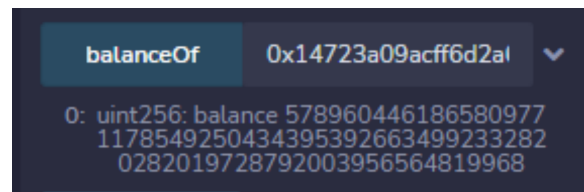

Fig. 35. Abnormal Transaction Result (1)

Fig. 36. Abnormal Transaction Result (2)

Under normal operations, the contract deducts the correct amount from the sender and adds it to the recipient. However, by crafting specific values for value and recipient addresses, an overflow is triggered, setting the internal calculated result to zero. This bypasses the balance check. When performing the transfer, the subtraction causes an underflow, leaving the sender's balance unchanged. Meanwhile, the value is transferred in full, causing a significant increase in the recipient's balance.

(2) Secure Mode.
(i). Contract deployment and initial setup. Figure 37 shows the deployment of the contract in secure mode.

(ii).Attack Results. As shown in Figure 37 and Figure 38, modifying the code to use SafeMath, the overflow operation fails, an exception is thrown, and execution is halted.

Fig. 37. Secure Contract Deployment Using SafeMath

Fig. 38. Attack Outcome Using SafeMath

(iii). Result analysis. SafeMath uses `assert()` statements to check for arithmetic anomalies. Any attempt to exceed the bounds of integer operations will revert the transaction, effectively preventing overflow vulnerabilities.

5 Conclusion

To mitigate reentrancy vulnerabilities caused by unbounded gas forwarding in `call()` and the malicious exploitation of fallback functions, employing `transfer()` for fund transfers offers a more secure alternative due to its fixed gas stipend. In the case of integer overflows, introducing arithmetic checks and adopting libraries such as SafeMath—with `assert()` to raise exceptions—effectively prevents such errors. Whether vulnerabilities stem from flaws in smart contract logic or the underlying EVM execution, ensuring smart contract security demands a holistic and multi-layered approach. This encompasses secure contract design, rigorous implementation and testing, robust regulatory oversight, and responsible user behavior. Ultimately, safeguarding smart contracts requires a collaborative effort among developers, auditors, users, and regulators to uphold both technical rigor and ethical standards throughout the contract lifecycle.

References

[1] Tianyuan Hu, Zecheng Li, Bixin Li, and Qihao Bao. A survey on smart contract security and privacy. *Journal of Computer Science and Technology*, 44(12):2485–2514, 2021.

[2] Sowon Jeon, Gilhee Lee, Hyoungshick Kim, and Simon S Woo. Design and evaluation of highly accurate smart contract code vulnerability detection framework. *Data Mining and Knowledge Discovery*, 38(3):888–912, 2024.

[3] Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal*, 10(14):12178–12185, 2023.

[4] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. Cobra: interaction-aware bytecode-level vulnerability detector for smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1358–1369, 2024.

[5] Zekai Liu and Xiaoqi Li. Sok: Security analysis of blockchain-based cryptocurrency. *arXiv preprint arXiv:2503.22156*, 2025.

[6] Jiuyang Bu, Wenkai Li, Zongwei Li, Zeng Zhang, and Xiaoqi Li. Smartbugbert: Bert-enhanced vulnerability detection for smart contract bytecode. *arXiv preprint arXiv:2504.05002*, 2025.

[7] Zongwei Li, Wenkai Li, Xiaoqi Li, and Yuqing Zhang. Stateguard: Detecting state derailment defects in decentralized exchange smart contract. In *Companion Proceedings of the ACM Web Conference 2024*, pages 810–813, 2024.

[8] Xiaoqi Li, Yingjie Mao, Zexin Lu, Wenkai Li, and Zongwei Li. Scla: Automated smart contract summarization via llms and control flow prompt. *arXiv e-prints*, pages arXiv–2402, 2024.

[9] Tsung-Ting Kuo and Anh Pham. Quorum-based model learning on a blockchain hierarchical clinical research network using smart contracts. *International journal of medical informatics*, 169:104924–104933, 2023.

[10] Hemang Subramanian and Susmitha Subramanian. Improving diagnosis through digital pathology: Proof-of-concept implementation using smart contracts and decentralized file storage. *Journal of medical Internet research*, 24(3):34207–34224, 2022.

[11] Pian Qi, Diletta Chiaro, Fabio Giampaolo, and Francesco Piccialli. A blockchain-based secure internet of medical things framework for stress detection. *Information Sciences*, 628:377–390, 2023.

[12] Huanhuan Zou, Zongwei Li, and Xiaoqi Li. Malicious code detection in smart contracts via opcode vectorization. *arXiv preprint arXiv:2504.12720*, 2025.

[13] Yingjie Mao, Xiaoqi Li, Wenkai Li, Xin Wang, and Lei Xie. Scla: Automated smart contract summarization via llms and semantic augmentation. *arXiv preprint arXiv:2402.04863*, 2024.

[14] Xiaoqi Li et al. Hybrid analysis of smart contracts and malicious behaviors in ethereum. *Hong Kong Polytechnic University*, 2021.

[15] Xiaoqi Li, L Yu, and XP Luo. On discovering vulnerabilities in android applications. In *Mobile Security and Privacy*, pages 155–166. Elsevier, 2017.

[16] Zongwei Li, Xiaoqi Li, Wenkai Li, and Xin Wang. Scalm: Detecting bad practices in smart contracts through llms. *arXiv preprint arXiv:2502.04347*, 2025.

[17] Dechao Kong, Xiaoqi Li, and Wenkai Li. Characterizing the solana nft ecosystem. In *Companion Proceedings of the ACM Web Conference 2024*, pages 766–769, 2024.

[18] Yuanzheng Niu, Xiaoqi Li, Hongli Peng, and Wenkai Li. Unveiling wash trading in popular nft markets. In *Companion Proceedings of the ACM Web Conference 2024*, pages 730–733, 2024.

[19] Xiaoqi Li, Ting Chen, Xiapu Luo, and Chenxu Wang. Clue: towards discovering locked cryptocurrencies in ethereum. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1584–1587, 2021.

[20] Yishun Wang, Xiaoqi Li, Shipeng Ye, Lei Xie, and Ju Xing. Smart contracts in the real world: A statistical exploration of external data dependencies. *arXiv preprint arXiv:2406.13253*, 2024.

[21] Xiaobing Bi, Zhaofeng Ma, and Mingkun Xu. Research and implementation of blockchain smart contract security development technology. *Information Security and Communication Secrecy*, (12):63–73, 2018.

[22] Zekai Liu, Xiaoqi Li, Hongli Peng, and Wenkai Li. Gastrace: Detecting sandwich attack malicious accounts in ethereum. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 1409–1411. IEEE, 2024.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, pages 1–9, 2008.

[24] Shuo Wang. Research status and innovation trends of blockchain technology in the financial field. *Shanghai Finance*, (2):26–29, 2016.

[25] Jiuyang Bu, Wenkai Li, Zongwei Li, Zeng Zhang, and Xiaoqi Li. Enhancing smart contract vulnerability detection in dapps leveraging fine-tuned llm. *arXiv preprint arXiv:2504.05006*, 2025.

[26] Abba Garba. A detailed explanation of defi protocol smart contract vulnerabilities: 4 major categories and 38 cases, 2021. Accessed: 2025-04-29.

[27] Amara Khatri. Defi protocol dforce suffers reentrancy attack, $3.6 million lost, 2023. https://cryptodaily.co.uk/2023/02/defi-protocol-dforce-suffers-reentrancy-attack-3-6-million-lost. Accessed: 2023-04-30.

[28] Jinlei Sun, Song Huang, Changyou Zheng, Tingyong Wang, Cheng Zong, and Zhanwei Hui. Mutation testing for integer overflow in ethereum smart contracts. *Tsinghua Science and Technology*, 27(1):27–40, 2021.

[29] Openzeppelin. The era of consolidation: Ethereum's revitalization through disruption of new consensus, 2023. https://www.openzeppelin.com/. Accessed: 2023-04-26.