Sufficient condition to judge bug

Feng Xiao

May 2020

1 introduction

This document is the full version of the paper ***A Framework and Data Set for Bugs in Ethereum Smart Contracts***. In this document, we describe the causes, consequences, examples, and detect criteria (sufficient conditions) of the 49 smart contract bugs mentioned in the paper ***A Framework and Data Set for Bugs in Ethereum Smart Contracts***.

2 How to formulate testing standards?

In general, we formulate detect criteria based on the following three strategies:

- **Strategy 1**. The cause of the bug or the behavior that caused the bug. The detect criteria for most kinds of bugs are based on this strategy:
- The most common form of code for bugs. For some kinds of bugs, the behaviors that cause the bugs are vague or the causes of the bugs are common , so we formulate detect criteria based on this strategy. For example, the *public data* bug, it is difficult to determine which structure is the **secret** of the contract. To detect this kind of bugs, we can only infer from the most common situation. When the developer wants to specify a structure as **secret**, similar in other programming languages, developers generally use the *private* keyword to modify the structure, so we will use the *private* keyword as a sign to judge the *public data* bug (Similarly, the detect criteria for the *incorrect inheritance order* bug are also determined according to this strategy).
- Missed judgments and misjudgments in smart contract analysis tools. We have used several smart contract analysis tools to detect contracts that contain multiple bugs and know some programming patterns that can easily lead to misjudgment or missed judgment. To help developers improve the detection performance of analysis tools, we add some detect criteria based on this strategy.

The detect criteria we give is based on features, that is, if there are certain features in the contract, it can be considered that there is a kind of bug in the contract.

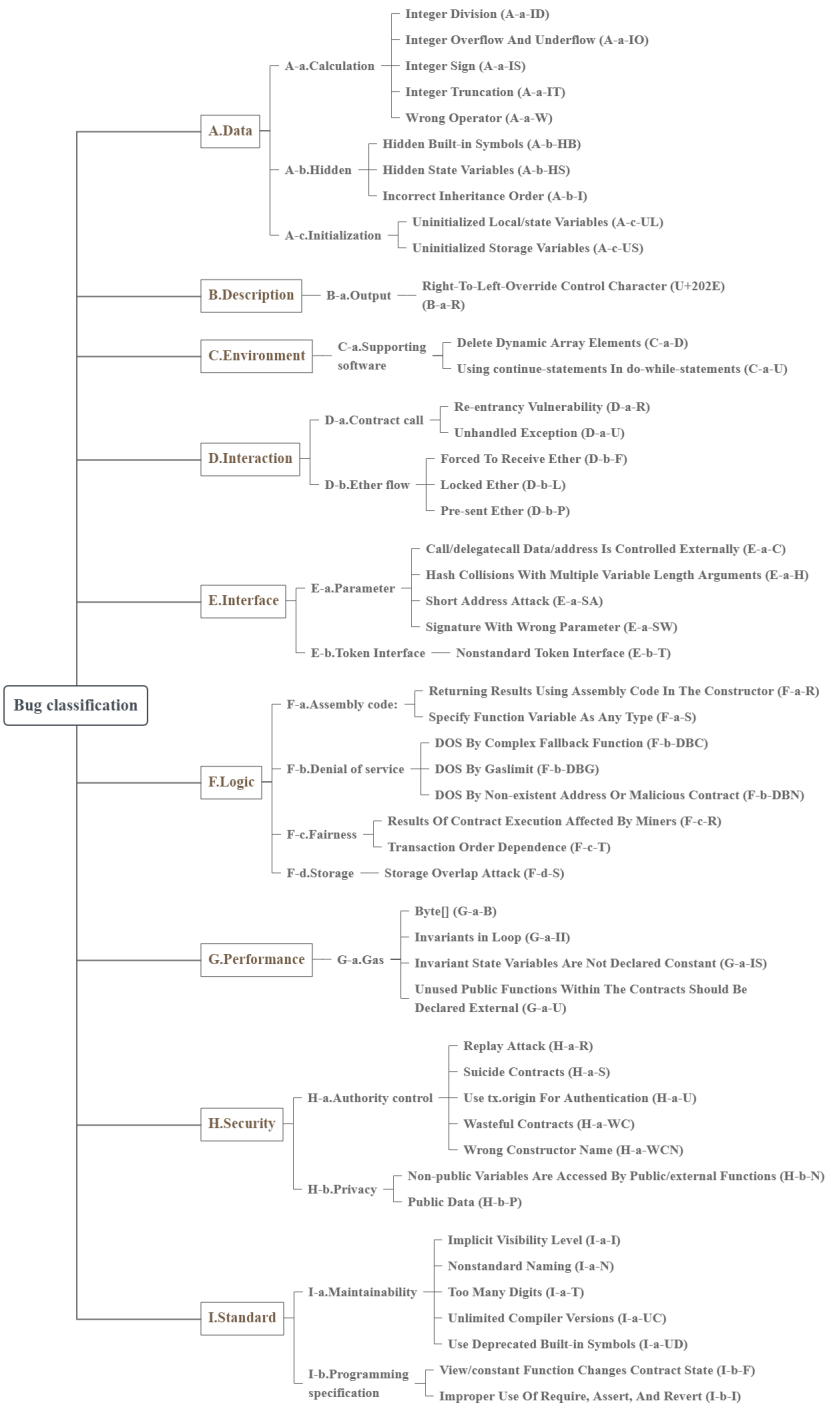


Figure 1: Smart contract bug classification

3 Smart contract bug classification

We cut and expand the *IEEE Standard Classification for Software Anomalies* to classify 49 kinds of smart contract bugs into 9 categories, as shown in Fig 1.

4 Details of various bugs

1. *Integer Division*:

   - *Cause*: All integer division results in *Solidity* are rounded down.
   - *Consequence*: This kind of bugs will cause a loss of accuracy in the calculation result. If the calculation result is used to represent the number of tokens (ethers), it will cause economic losses.
   - *Example*: Consider the smart contract in Fig 2, which is used to calculate employee salaries. In the 18th line of code, the values of the two variables involved in the division are given externally. When the calculation results of the two variables are not integers (eg., *DailyWage*: 100, *coefficient*: 3), It will make the employee's actual salary lower than the amount due.

```
1  pragma solidity 0.6.2;
2  contract getWageNumber {
3      uint256 public coefficient;
4      uint256 public DailyWage;
5      address public boss;
6      constructor() public{
7          DailyWage = 100;
8          coefficient = 3;
9          boss = msg.sender; }
10     modifier onlyOwner{
11         require(msg.sender == boss);
12         _; }
13     function setDailyWage(uint256 _wage) external onlyOwner{
14         DailyWage = _wage; }
15     function setCoefficient(uint256 _co) external onlyOwner{
16         coefficient = _co; }
17     function calculateWage(uint256 dayNumber) external view onlyOwner returns (uint256) {
18         uint256 baseWage = DailyWage / coefficient; //integer division
19         return baseWage * dayNumber; } }
```

Figure 2: An contract contains an *integer division* bug

   - *detect criteria*: This kind of bugs exists when the contract meets any of the following characteristics:
     - The result of dividing two numeric constants (the dividend is not the multiple of the divisor) is assigned to an integer variable.
     - The result of dividing two integer variables is assigned to an integer variable, but the result is not necessarily an integer.

2. *Integer Overflow And Underflow*:

   - *Cause*: When the result exceeds the boundary value, the result will overflow or underflow.

- *Consequence*: This kind of bugs will cause erroneous calculation results. If these calculation results are used to represent the amount of tokens (ethers), it will cause economic losses.
- *Example*: Consider the smart contract in Fig 3, assuming that the value of *sellerBalance* has been accumulated to $2^{256} - 1$, if the user calls the *add* function and the given parameter (*value*) is 1, then the value of *sellerBalance* will exceed the upper limit ($2^{256} - 1$), the *sellerBalance* value returned last is 0.

```
1  pragma solidity 0.6.2;
2  contract Overflow {
3      uint private sellerBalance=0;
4    constructor() public{}
5      function add(uint value) public returns (uint256){
6          sellerBalance += value; //may overflow
7          return sellerBalance;
8      } }
```

Figure 3: An contract contains an *integer overflow and underflow* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The maximum/minimum value that can be generated by the operands involved in integer operations will exceed the storage range of the variable responsible for storing the operation result.
  - The contract does not verify whether the calculation result that may overflow/underflow is correct.

3. *Integer Sign*:

- *Cause*: In *Solidity*, Converting *int* type to *uint* type (and vice versa) may produce incorrect results.
- *Consequence*: The consequences of this bug are the same as the *integer overflow and underflow* bug.
- *Example*: Consider the smart contract in Fig 4, if the attacker calls the *withdrawOnce* function and specifies that the value of *amount* is a negative number, then this call will pass the check (line 10) and transfer out the ethers that exceed the limit (1 ether).
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - Force conversion of an *int* variable that may be negative to a *uint* variable.
  - The contract does not check whether this *int* variable is negative.

4. *Integer Truncation*:

- *Cause*: Casting a long integer variable into a short integer variable may result in a loss of accuracy (eg. *uint*256 *to uint*8).

```
1  pragma solidity 0.6.2;
2  contract signednessError{
3      mapping(address => bool) public transferred;
4      address public owner;
5      constructor() public payable{
6          owner = msg.sender;
7          require(msg.value > 0 && msg.value % 1 ether == 0);
8      }
9      function withdrawOnce (int amount) public {
10         if ( amount > 1 ether || transferred [msg.sender]) {
11             revert() ;
12         }
13         msg.sender.transfer(uint(amount));
14         transferred [msg.sender] = true ;
15     }
16 }
```

Figure 4: A contract that contains a *integer sign* bug

- *Consequence*: The consequences of this bug are the same as the *integer division* bug.
- *Example*: Consider the smart contract in Fig 5, the code on line 8 contains an *integer truncation* bug. When the value of *msg.value* is greater than the maximum value that the *uint32* type can hold, the result of the forced conversion will produce a loss of accuracy.

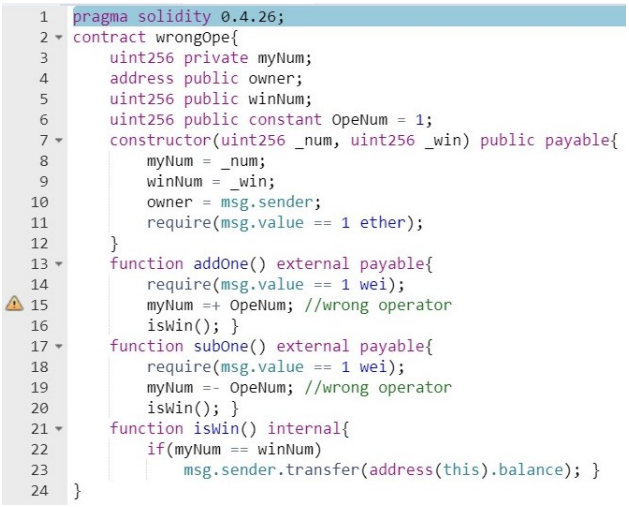
```
1  pragma solidity 0.6.2;
2  contract truncationError{
3      mapping(address => uint32) public balances;
4      constructor() public{}
5      function receiveEther() public payable{
6          //truncation Error
7          require(balances[msg.sender] + uint32(msg.value) >= balances[msg.sender]);
8          balances[msg.sender] += uint32(msg.value);
9      } }
```

Figure 5: A contract that contains a *integer sign* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There is an operation in the contract to forcibly convert a long integer variable into a short integer variable.
  - There is no operation in the contract to check the value of the long integer variable before conversion.

5. *Wrong Operator*:

- *Cause*: Users can use =+ and =- operators in the integer operation without compiling errors (up to and including version 0.4.26).
- *Consequence*: The consequences of this bug are the same as the *integer overflow and underflow* bug.
- *Example*: Consider the smart contract in Fig 6, the user can adjust the value of *myNum* by calling *addOne*/*subOne* function. When *myNum* and *WinNum* are equal, the user gets all the ethers of the



```
1  pragma solidity 0.4.26;
2  contract wrongOpe{
3      uint256 private myNum;
4      address public owner;
5      uint256 public winNum;
6      uint256 public constant OpeNum = 1;
7      constructor(uint256 _num, uint256 _win) public payable{
8          myNum = _num;
9          winNum = _win;
10         owner = msg.sender;
11         require(msg.value == 1 ether);
12     }
13     function addOne() external payable{
14         require(msg.value == 1 wei);
15         myNum =+ OpeNum; //wrong operator
16         isWin(); }
17     function subOne() external payable{
18         require(msg.value == 1 wei);
19         myNum =- OpeNum; //wrong operator
20         isWin(); }
21     function isWin() internal{
22         if(myNum == winNum)
23             msg.sender.transfer(address(this).balance); }
24 }
```

Figure 6: A contract contains a *wrong operator* bug

contract. But because the developers write the wrong operators (line 15, 19), the value of *myNum* will not change.

- *detect criteria*: This kind of bugs exists when the following characteristics are present in the contract:
  - There is a =+ or =- operator in the contract.

6. *Hidden Built-in Symbols*:

   - *Cause*: When a variable with the same name as a built-in symbol exists, the built-in symbol is hidden.
   - *Consequence*: This kind of bugs will make the built-in symbol hidden, which may cause the built-in symbol to become invalid and ultimately affect the function of the contract.
   - *Example*: Consider the smart contract in Fig 7, the function declared on line 7 causes the built-in method *gasleft()* to be hidden, which makes the condition judgment part on line 12 permanently return *false*.
   - *detect criteria*: This kind of bugs exists when the following characteristics are present in the contract:
     - The name of the identifier defined by the developer is the same as the built-in symbol of the programming language.

7. *Hidden State Variables*:

   - *Cause*: Variables in subclasses will override variables of the same name in the base class.

```
pragma solidity 0.6.2;
contract HiddenBuiltinSymbols {
    address public owner;
    constructor() public payable{
        owner = msg.sender;
        require(msg.value > 0); }
    function gasleft() external returns(uint256){
        return 10; }
    function withdraw() external{
        require(msg.sender == owner);
        //always false
        if (gasleft() >= 2300 )
            msg.sender.transfer(address(this).balance);  } }
```

Figure 7: A contract contains a *wrong operator* bug

- *Consequence*: This kind of bugs will cause the state variable of the same name in the base contract to be hidden, which may affect the function of the subclass contract.
- *Example*: Consider the smart contract in Fig 8, the *owner* variable in the subclass *SonContract* overrides the variable of the same name in the base class, which will cause the *onlyOwner* in the base class to be invalid.

```
pragma solidity 0.5.16;
contract FatherContract{
    address owner;
    modifier onlyOwner{
        require(owner == msg.sender);
        _;
    }
}
contract SonContract is FatherContract{
    address owner;
    constructor() public payable{
        owner = msg.sender;}
    function withdraw() external onlyOwner{
        msg.sender.transfer(address(this).balance);}
}
```

Figure 8: A contract contains a *hide state variables* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The subclass contract and the base class contract contain state variables of the same name.
  - The function of the base contract depends on the hidden state variables.

8. *Incorrect Inheritance Order*:

- *Cause*: When multiple inheritances occur, two (or more) base contracts may overwrite the structure defined by each other.
- *Consequence*: This kind of bugs may cause the subclass contract's function to be inconsistent with the developer's expectations.

- *Example*: Consider the smart contract in Fig 9. Generally speaking, the contracts recently written by developers are more in line with current needs. So for the *getNum* function, the *getNum* function in contract *D* is most in line with the developer's expectations. When specifying the multiple inheritance order of contract *E*, the *getNum* function in contract *D* is overwritten by the function of the same name in contract *B*, which results in the abnormal function of contract *E*.

```
pragma solidity 0.5.16;

contract A{
    uint256 public num;
    function getNum() view public returns(uint256){
        return num;
    }
}
contract B is A{
    constructor(uint256 _num) public{
        num = _num;
    }
}
contract C is A{
    function setNum() public{
        num += 10;
    }
}
contract D is C{
    function getNum() view public returns(uint256){
        return num + 10;
    }
}
contract E is D, B{
    address public owner;
    constructor() public{
        owner = msg.sender;
    }
}
```

Figure 9: A contract contains a *incorrect inheritance order* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There is multiple inheritance in the contract.
  - There are structures with the same name in multiple inherited base contracts (eg., function, variable, modifier).
  - The base contract recently written by the developer is specified after the base contract written earlier in the priority of the *multiple inheritance*.

9. *Uninitialized Local/state Variables*:

- *Cause*: Uninitialized local/state variables will be given default values (eg., the default value of the *address* variable is *0x0*, sending ethers to this address will cause ethers to be destroyed).
- *Consequence*: This kind of bugs will increase the probability of developers making mistakes, which may eventually lead to the contract's function not being consistent with the developers' expectations.

- *Example*: Consider the smart contract in Fig 10. Since the developer did not give the initial value of the *number* (line 11), the number of times the *giveMeMoney* function was called cannot be recorded in the *Record* event.

```
1  pragma solidity 0.6.2;
2  contract uninitStateLocalVar{
3      address payable payee;
4      uint256 public _order;
5      event Record(address indexed _donors, uint256 _number, uint256 _money);
6      constructor() public{
7          _order = 0;
8          payee = msg.sender;
9      }
10     function giveMeMoney() external payable{
11         uint256 number;
12         _order += 1;
13         require(msg.value > 0);
14         emit Record(msg.sender, number, msg.value);
15     }
16     function withdrawDonation() external{
17         payee.transfer(address(this).balance);
18     }
19 }
```

Figure 10: A contract contains a *uninitialized local/state variables* bug

- *detect criteria*: This kind of bugs exists when the following characteristics are present in the contract:
  - The developers do not initialize the local/state variables in the contract.

10. *Uninitialized Storage Variables*:

- *Cause*: The uninitialized storage variable serves as a reference to the first state variable, which may cause the state variable to be inadvertently modified (up to and including version 0.4.26).
- *Consequence*: This kind of bugs may cause key state variables to be rewritten inadvertently, and eventually, the function of the contract will be affected.
- *Example*: Consider the smart contract in Fig 11, when the user calls the function *func*, the owner will be re-assigned to 0x0 (the default storage location of the variable st is **storage**).
- *Detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - The developers do not initialize the *storage* variables in the contract.

11. *Right-To-Left-Override Control Character*:

- *Cause*: Using *U+202E* characters will cause the output string to be inverted.

```solidity
pragma solidity 0.4.26
contract Uninitialized{
    address owner = msg.sender;
    struct St{
        uint a;
    }
    function func() {
        St st;
        st.a = 0x0; //owner is override to 0.
    }
}
```

Figure 11: A contract contains a *uninitialized storage variables* bug

- *Consequence*: This kind of bugs will cause string inversion (including source code, comments, or event output).
- *Example*: Consider the code in Fig 12. The actual content of the parameters of the function *checkAndTransferPrize* is the reverse of the displayed content.

```solidity
function guess(uint n) payable public{
        require(msg.value == 1 ether);
        uint p = address(this).balance;
        checkAndTransferPrize(/*The prize*/p , n/*guessed number*/
                /*The user who should benefit */,msg.sender);
}
```

Figure 12: A contract contains a *right-to-left-override control character* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - The source code of the contract (including comments) contains *U+202E* characters.

12. *Delete Dynamic Array Elements*:

- *Cause*: Deleting dynamic array elements does not automatically shorten the length of the array and move the array elements.
- *Consequence*: This kind of bugs will cause the deleted elements in the array to be set to the default value, which will increase the probability of developers making mistakes and may eventually affect the function of the contract.
- *Example*: Consider the smart contract in Fig 13. Assume that the contract owner first calls the *deletePartner* function to delete an address in the array, which will cause this element to be set to *0x0*, and then call the *sendEther* function, which will cause at least one ether to be sent to address *0x0*, and all sent to the address *0x0* ethers will be destroyed.

```
pragma solidity 0.6.2;
//I will send 1 ether to each of my business partners.
contract myBonus{
    address public owner;
    address payable[] public myPartners;
    constructor() public{
        owner = msg.sender;
    }
    modifier onlyOwner{
        require(msg.sender == owner);
        _;
    }
    function depositOnce() external payable onlyOwner{
        require(msg.value == 1 ether);
    }
    function addNewPartner(address payable _friend) external onlyOwner{
        myPartners.push(_friend);
    }
    function deletePartner(address _badGuy) external onlyOwner{
        uint256 _length = myPartners.length;
        for(uint256 i = 0; i < _length; i++){
            if(myPartners[i] == _badGuy)
                delete myPartners[i];
        }
    }
    function sendEther() external onlyOwner{
        uint256 _length = myPartners.length;
        for(uint256 i = 0; i < _length; i++){
            myPartners[i].transfer(1 ether);
        }
    }
}
```

Figure 13: A contract contains a *delete dynamic array elements* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - In the source code, there are some operations to delete the array elements by using the *delete-statement*.
  - There is no statement in the contract to move subsequent elements in the array.

13. *Using continue-statements In do-while-statements*:

- *Cause*: Executing the *continue*-statements in the *do-while*-statements causes the condition decision statement to be skipped once (up to and including version 0.4.26).
- *Consequence*: This kind of bugs will cause the condition judgment part of the *do-while-statement* to be skipped once, which will cause the contract's function to be inconsistent with the developer's expectations, and in extreme cases will cause an infinite loop.
- *Example*: Consider the function in Fig 14, assuming that *_own* is not the last element of the array, then this *do-while* loop will become an infinite loop.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:

```
function deleteOwner(address _own) external onlyOwner{
    uint256 i = 1;
    uint256 _length = owners.length;
    do{
       if(owners[i] != _own || owners[i] == address(0x0)){
           i++;
           continue;
       }
       else{
           delete owners[i];
           i++;
       }
    }while( i != _length);
}
```

Figure 14: A contract contains a *using continue-statements in do-while-statements* bug

- Developers use *Solidity* before version 0.5.0 to write smart contracts.
- The *continue-statement* and *do-while-statement* are used.
- The *continue-statement* exists in the *do-while-statement*.
- The *continue-statement* in the *do-while-statement* is reachable.

14. *Re-entrancy Vulnerability*:

- *Cause*: When the *call*-statement is used to call other contracts, the callee can call back the caller and enter the caller again.
- *Consequence*: This kind of bugs is one of the most dangerous smart contract bugs, which will cause the contract balance (ethers) to be stolen by attackers.
- *Example*: We use an example to illustrate the *re-entrancy vulnerability*. In Fig 15, the contract *Re* is a contract with a *re-entrancy vulnerability*, and the *balance* variable is a map used to record the correspondence between the address and the number of tokens. The attacker deploys the contract *Attack*, and the value of the parameter *_reAddr* is set to the address of the contract *Re*. In this way, the *re*

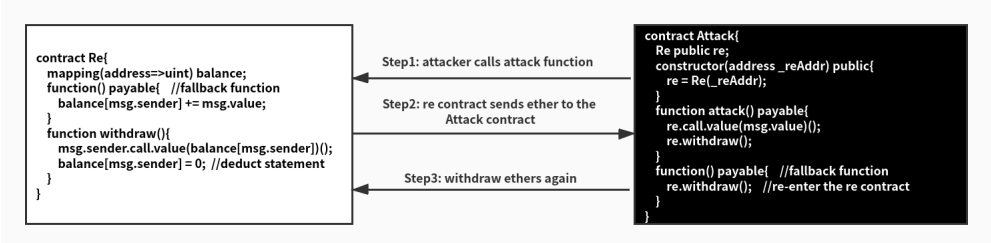


Figure 15: An example of *re-entrancy vulnerability*

variable becomes an instance of the contract *Re*. Then,

  - *Step 1*: The attacker calls the *attack* function to deposit ethers into the contract *Re* and then calls the *Re.withdraw* function to retrieve the deposited ethers.
  - *Step 2*: The contract *Re* executes the *withdraw* function and uses a *call*-statement to send ethers to the contract *Attack*. At this time, the power of control is transferred to the contract *Attack*, and the contract *Attack* responds to the transfer using the *Attack.fallback* function.
  - *Step 3*: The *Attack.fallback* function calls the *Re.withdraw* function to withdraw the ethers again. Therefore, the statement (*deduct-statement*) deducting the number of tokens held by the contract *Attack* will not be executed.
- *Detect criteria*: When there are the following characteristics in the contract, it will cause the *reentrancy bug*:
  - The *call-statement* is used to send ethers.
  - The amount of *gas* to be carried is not specified.
  - No callee's response function is specified.
  - Ethers are transferred first and callee's balance is deduced later.

15. *Unhandled Exception*:

- *Cause*: The contract can use low-level call statements such as *send*, *call*, and *delegatecall* to interact with other addresses. When a call made using these low-level call statements is abnormal, the call is not terminated and rollback, only *false* is returned.
- *Consequence*: This kind of bugs will cause the contract to be unable to know and handle the exceptions generated during the call, which may affect the function of the contract.
- *Example*: Consider the smart contract in Fig 16. When the contract owner (*bankOwner*) calls the *refundAll* function, he cannot know which users successfully received the refund (the code on line 24 uses the *call-statement* to transfer ethers and does not receive the return value of the *call-statement*).
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There are low-level call statements in the contract (without comments).
  - The contract does not receive the return value or receives the return value but does not check the return value.

16. *Forced To Receive Ether*:

- *Cause*: An attacker can force ethers to be sent to an address through self-destructing contracts or mining.

```
pragma solidity 0.6.2;
contract xfBank{
    mapping(address => uint256) public ledger;
    address[] public user;
    address public bankOwner;
    constructor() public{
        bankOwner = msg.sender;
    }
    function deposit() external payable{
        require(msg.value > 0);
        ledger[msg.sender] += msg.value;
        user.push(msg.sender);
    }
    function withdraw(uint256 _money) external{
        require(ledger[msg.sender] >= _money);
        msg.sender.transfer(_money);
    }
    function refundAll() external{
        require(msg.sender == bankOwner);
        uint256 _length = user.length;
        for(uint256 i = 0 ; i < _length; i++){
            uint256 money = ledger[user[i]];
            ledger[user[i]] = 0;
            user[i].call.value(money)("");
        }
    }
}
```

Figure 16: A contract contains a *unhandled exception* bug

- *Consequence*: An attacker can forcibly send ethers to a contract. If the contract's function depends on the contract's balance being at a certain value, the attacker can use this error to affect or destroy the contract's function.
- *Example*: Considering the smart contract in Fig 17, the attacker can forcibly send a little ethers (eg., 1 Wei) to the contract *BobAnswer*,which will cause the conditional judgment part of the 10th line of code to return *false* forever.

```
pragma solidity 0.6.2;
contract BobAnswer{
    uint256 private answer; //miners can see the answer's value
    address public owner;
    constructor(uint256 _answer) public{
        owner = msg.sender;
        answer = _answer;
    }
    function getAnswer() external payable returns(uint256){
        require(address(this).balance % 1 ether == 0);
        require(msg.value > 0);
        return answer;
    }
    function withdraw() external{
        require(msg.sender == owner);
        msg.sender.transfer(address(this).balance);
    }
}
```

Figure 17: A contract contains a *forced to receive ether* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:

  - The function of the contract is related to whether the balance of the contract is at a certain value.

17. *Locked Ether*:

    - *Cause*: If the contract can receive ethers, but cannot send ethers, the ethers in the contract will be permanently locked.
    - *Consequence*: This kind of bugs will cause all ethers in the contract to be permanently locked and will cause economic losses.
    - *Example*: Considering the smart contract in Fig 18, all ethers transferred into this contract cannot be transferred out and will be permanently locked.

```
1  pragma solidity 0.6.2;
2  contract BadMarketPlace {
3      function deposit() external payable {
4          require(msg.value > 0);
5      }
6  }
```

Figure 18: A contract contains a *locked ether* bug

    - *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
      - At least one function in the contract is declared *payable*.
      - There is no statement in the contract to transfer out ethers, or the statement to transfer out ethers is unreachable.

18. *Pre-sent Ether*:

    - *Cause*: Malicious users can send ethers to the address of the contract before the contract is deployed. If the function of the contract depends on the balance of the contract, then the *pre-sent ether* may affect the function of the contract.
    - *Consequence*: The consequences of this bug are the same as the *forced to receive ether* bug.
    - *Example*: Consider the smart contract in Fig 19. If the attacker knows the address where the contract will be deployed in advance, a small amount of ethers (eg., 1 *wei*) can be sent in advance, which will cause the conditional judgment part of the line 10 code to permanently return *false*.
    - *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
      - The function of the contract is related to whether the balance of the contract is zero.

19. *Call/delegatecall Data/address Is Controlled Externally*:

```
pragma solidity 0.6.2;
contract PreSentEther{
    address public owner;
    uint256 public money;
    constructor() public{
        owner == msg.sender;
        money = 0;
    }
    function depositOnce() external payable{
        require(address(this).balance == 0);
        money += msg.value;
    }
    function withdraw() external{
        require(msg.sender == owner);
        msg.sender.transfer(money);
    }
}
```

Figure 19: A contract contains a *pre-sent ether* bug

- *Cause*: Contracts can call functions of other contracts. If the call data or address is controlled externally, the attacker can arbitrarily specify the call address, call function and parameters.
- *Consequence*: Attackers can use this kind of bugs to make the contract perform functions that developers do not want to perform. This kind of bugs will affect the function of the contract and may cause economic losses.
- *Example*: Consider the smart contract in Fig 20. Assume that the attacker has deployed the attack contract, and this contract contains a function named *withdrawAll*. The function of the *withdrawAll* is to transfer out all the ethers in the contract to the attacker. Then the attacker can call the *forward* function, the parameters are given as the address of the attack contract and the signature of the *withdrawAll* function, so that the *withdrawAll* function is executed in the contract *Proxy*, thereby sending all ethers of the contract proxy to the attacker.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The parameters (address or data) of the *call/delegatecall-statement* are given by the external users.
  - The contract does not check these externally given parameters.

20. *Hash Collisions With Multiple Variable Length Arguments*:

- *Cause*: Because *abi.encodePacked()* packs all parameters in order, regardless of whether the parameters are part of an array, the user can move elements within or between arrays. As long as all elements are in the same order, *abi.encodePacked()* will return the same result.

```
1  pragma solidity 0.6.2;
2  contract Proxy {
3    address public owner;
4    constructor() public payable{
5      owner = msg.sender;
6      require(msg.value > 0);
7    }
8    modifier onlyOwner{
9        require(msg.sender == owner);
10       _;
11   }
12   function forward(address callee, bytes memory data) public {
13     callee.delegatecall(data);
14   }
15   function withdraw() external{
16       require(msg.sender == owner);
17       msg.sender.transfer(address(this).balance);
18   }
19 }
```

Figure 20: A contract contains a *call/delegatecall data/address is controlled externally* bug

- *Consequence*: This kind of bugs will cause different parameters to produce the same hash value, which may affect the function of the contract.
- *Example*: Consider the smart contract in Fig 21, the function addUsers takes two arrays of the same type as parameters and uses the *abi.encodePacked()* (line 11) package two arrays. Suppose there are two calls to the addusers function. The parameters of the first call are: (['0x1 ','0x2'], ['0x3 '],'1'), and the parameters of the second call are (['0x1 '], ['0x2','0x3 '], 1). These two calls will produce the same hash value, so the contract's function is not consistent with the developer's expectations.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - Use *abi.encodePacked()* to pack the parameters of the hash function.
  - The parameters of *abi.encodePacked()* contain multiple arrays of the same type.

21. *Short Address Attack*:

- *Cause*: When Ethereum packs transaction data if the data contains the address type and the length of the address type is less than 20 bits, subsequent data will be used to make up the length of the address type.
- *Consequence*: This kind of bugs may cause the attacker to manipulate tokens (ethers) equivalent to several times the number he requested.
- *Example*: We use an example to illustrate *short address attacks*.

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

Figure 21: A contract contains a *hash collisions with multiple variable length arguments* bug

– *Step 1*: *Tom* deploys token contract A on Ethereum, which contains the *sendcoin* function. The code of *sendcoin* is shown in Fig 22.

```
function sendCodin(address _to, uint256 _amount) returns(bool){
    if(balance[msg.sender] < _amount)
        return false;
    //using safemath for uint256
    balance[msg.sender] = balance[msg.sender].sub(_amount);
    balance[_to] = balance[_to].add(_amount);
    Tranfer(msg.sender, _to, _amount);
    return true;
}
```

Figure 22: Objective function of *short address attack*

– *Step 2*: *Jack* buys 100 tokens of contract A, then registers for an Ethereum account with the last two digits zero (eg. 0x1234567890 1234567890123456789012345678 00).
– *Step 3*: *Jack* calls the function *SendCoin* with the given parameters, *_to*: 0x1234567890123456789012345678901234567800 (missing last two digits 0), *_amount*: 50.
– *Step 4*: The value of *_amount* is less than 100, so it passes the check. However, because the bits of *_to* is insufficient, the first two bits (0) of *_amount* will be added to the *_to* when the transaction data is packed. Therefore, to make up for the length of *_amount*, the Ethereum virtual machine will add 0 to the last two bits. In the end, the value of *_amount* is expanded by four times.

- *Detect criteria*: When there are the following characteristics in the contract, it will cause the *short address attack*:
  - The contract uses a function to transfer ethers or tokens.
  - The number of tokens (ethers) and the address for receiving tokens (ethers) are provided by external users.
  - There is no operation to check the length of the received tokens (ethers) address in the function.

22. *Signature With Wrong Parameter*:

- *Cause*: When the parameters of the *ecrecover()* are wrong, the *ecrecover()* will return 0x0.
- *Consequence*: This kind of bugs will allow the attacker to pass the authentication and then the attacker can manipulate the token (ethers) held by the *0x0* address.
- *Example*: Considering the smart contract in Fig 23, when the attacker gives the wrong parameters (v, r, s) and the value of the specified parameter *_id* is *0x0*, the attacker can pass the identity verification (Line 10), which eventually leads to the ethers in the contract are destroyed.
- *Detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There is an operation in the contract that uses the *ecrecover()* to calculate the public key address.
  - The contract does not deal with the case where the *ecrecover()* returns *0x0*.

```
1  pragma solidity 0.6.2;
2  contract SigWrongPara{
3      bytes32 private idHash;
4      constructor() public payable{
5          require(msg.value > 0);
6          idHash = keccak256(abi.encode(msg.sender));
7      }
8      function getMyMoney(address payable _id,  uint8 v,
9        bytes32 r, bytes32 s) external returns(bool){
10         if(_id != ecrecover(idHash, v, r, s))
11             return false;
12         _id.transfer(address(this).balance);
13         return true;
14     }
15 }
```

Figure 23: A contract contains a *signature with wrong parameter* bug

23. *Nonstandard Token Interface*:

- *Cause*: Token contracts that do not meet *ERC20*, *ERC721* and other token standards may have problems when interacting with other contracts.

- *Consequence*: This kind of bugs may cause the interaction between the contract and other token contracts to be abnormal, thereby affecting the function of the contract.
- *Example*: Consider the function in Fig 24, when the function fails to execute, the function uses the *require-statement* (line 5) to throw an exception instead of returning *false*. Therefore, the token contract interacting with the contract cannot check whether the function is executed normally by checking the return value of the function.

```
1  /*
2  Raises an exception in a function that should return a Boolean value
3  */
4  function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
5      require(_value <= tokenAllowance[_from][msg.sender]);
6      tokenAllowance[_from][msg.sender] -= _value;
7      _transfer(_from, _to, _value);
8      return true;
9  }
```

Figure 24: A function contains a *nonstandard token interface* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Developers do not follow the provisions of the token standard (eg., *ERC20*) to develop token contracts.

24. *Returning Results Using Assembly Code In The Constructor*:

- *Cause*: Using assembly code return values in the constructor can make the contract deployment process inconsistent with developer expectations.
- *Consequence*: This kind of bugs will cause the execution effect of the constructor to be inconsistent with the expectation and may affect the function of the contract.
- *Example*: Consider the smart contract in Fig 25, and the code in the 6th line will make the deployment process of the contract not as expected. In fact, the contract deployed to the blockchain is equivalent to the contract shown in Fig 26.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Use assembly statement return values in the constructor.

25. *Specify Function Variable As Any Type*:

- *Cause*: Function variables can be specified as any type through assembly code (up to and including version 0.5.16).
- *Consequence*: This kind of bugs will cause the function variable to be assigned to any type, thus bypassing the necessary type checking. This will increase the probability of developers making mistakes and may affect the functionality of the contract.

```
pragma solidity 0.6.2;
contract HoneyPot {
    bytes internal constant ID = hex"60203414600857005B60008080803031335AF100";
    constructor () public payable {
        bytes memory contract_identifier = ID;
        assembly { return(add(0x20, contract_identifier), mload(contract_identifier)) }
    }
    function withdraw() public payable {
        require(msg.value >= 1 ether);
        msg.sender.transfer(address(this).balance);
    }
}
```

Figure 25: A contract contains a *returning results using assembly code in the constructor* bug

```
contract HoneyPot {

    function () public payable {
        if (msg.value == 32) {
            msg.sender.transfer(address(this).balance);
        }
    }
}
```

Figure 26: Deploy the actual effect of the contract in Fig 25

- *Example*: Consider the function in Fig 27, the 5th line of code forcibly assigns the variable *func.f* to another type, which bypasses the type check.

```
function breakIt() public payable {
    require(msg.value != 0, 'send funds!');
    Func memory func;
    func.f = frwd;
    assembly { mstore(func, add(mload(func), callvalue)) }  //wrong
    func.f();
}
```

Figure 27: A function contains a *specify function variable as any type* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Use the *mstore* instruction and the parameters of the *mstore* instruction contain function variables.

26. *DOS By Complex Fallback Function*:

- *Cause*: If the execution of the *fallback* function consumes more than 2300 *gas*, sending ethers to the contract using *transfer-statement* or *send-statement* may fail.

- *Consequence*: This kind of bug will cause the contract to be unable to receive some externally transferred ethers (the ethers sent using the *transfer-statement* and *send-statement*), which may cause economic losses.
- *Example*: Consider the function in Fig 28. When the *payer* array is large, the cost of executing the *fallback* function will exceed 2300 *gas*, which will make it impossible to transfer ethers through *transfer-statement* or *send-statement*.

```
1   //If you paid, record that you paid.
2   /*
3   In the worst case, this will traverse the entire array,
4   and when the array size is large, it will consume a large
5   amount of gas.
6   */
7   fallback() external payable{
8       require(msg.value > 0);
9       for(uint256 i = 0; i < payer.length; i++){
10          if(msg.sender == payer[i])
11              money[i] += msg.value;
12      }
13  }
```

Figure 28: A function contains a *DOS by complex fallback function* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Executing the fallback function consumes more than 2300 *gas*.

27. *DOS By Gaslimit*:

- *Cause*: There is an attribute in the blocks of Ethereum, *gaslimit*, which specifies the upper limit of *gas* consumed by all transactions in the block. When a transaction consumes too much *gas*, the transaction may be refused to be packaged.
- *Consequence*: This kind of bugs causes the transaction to be refused to be packaged, which lead to the failure of contract invocation.
- *Example*: Consider the function in Fig 29, when the length of the element array is greater than 255, it will cause the loop (line 3) to become an infinite loop. Even if enough *gas* is paid, the call containing this loop will fail because it exceeds the limit of the *gaslimit* attribute.

```
1   function addOne() public onlyOwner{
2       uint256 _length = element.length;
3       for(uint8 i = 0; i < _length; i++)
4           element[i] += 1;
5   }
```

Figure 29: A function contains a *DOS by gaslimit* bug

- *detect criteria*: This kind of bugs exists when the contract meets any of the following characteristics:
  - There may be an infinite loop in the contract.
  - There is an operation to delete an array using the *delete-statement* in the contract, and the size of the deleted array may be large.

28. *DOS By Non-existent Address Or Malicious Contract*:

- *Cause*: When the address that interacts with the contract does not exist, or the callee contract has an exception, the call will fail.
- *Consequence*: This kind of bug will affect the function of the contract, and in extreme cases may cause the contract's function to be completely invalid.
- *Example*: Consider the smart contract in Fig 30. If the *owner* variable on line 7 is assigned the wrong value, then the contract cannot provide services correctly.

```
1  pragma solidity 0.6.2;
2  contract HardCodeAddress {
3      uint256 private rate;
4      uint256 private cap;
5      address public owner;
6      constructor() public  {
7          owner = 0x5aAeb6053F3E94C9b9A09f33669435E7Ef1BeAed;
8          rate = 0;
9          cap = 0;
10     }
11     function setRate(uint256 _rate) external  {
12         require(owner == msg.sender);
13         rate = _rate;
14     }
15     function setCap(uint256 _cap) external {
16         require (msg.sender == owner);
17         cap = _cap;
18     }
19 }
```

Figure 30: A function contains a *DOS by non-existent address or malicious contract* bug

- *detect criteria*: This kind of bugs exists when the contract meets any of the following characteristics:
  - The contract function depends on the return value of the external contract.
  - Or the contract uses the *transfer-statement* in the loop to transfer out ethers to the external address.

29. *Results Of Contract Execution Affected By Miners*:

- *Cause*: The miner can control the attributes related to mining and blocks. If the functions of the contract depend on these attributes, the miner can interfere with the functions of the contract.

- *Consequence*: This kind of bugs will cause miners to gain a competitive advantage, which makes the contract's function inconsistent with the developer's expectations.
- *Example*: Consider the function in Fig 31, the miner can control the mining time so that the condition judgment code of the second line of code is always *true* (eg., given the parameter *yourGuess* is *true*, and then only mine at odd times).

```
1  function oddOrEven(bool yourGuess) external payable {
2      if (yourGuess == now % 2 > 0) {
3          uint fee = msg.value / 10;
4          msg.sender.transfer(msg.value * 2 - fee);
5      }
6  }
```

Figure 31: A function contains a *results of contract execution affected by miners* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - Special variables related to block attributes or mining attributes are used in the contract.
  - The function of the contract is related to the value of these special variables.

30. *Transaction Order Dependence*:

- *Cause*: Miners can decide which transactions are packaged into the blocks and the order in which transactions are packaged. The current main impact of This kind of bugs is the *approve* function in the *ERC20* token standard.
- *Consequence*: This kind of bugs will enable miners to influence the results of transaction execution. If the results of the previous transactions will have an impact on the results of the subsequent transactions, miners can influence the results of transactions by controlling the order in which the transactions are packaged.
- *Example*: The *approve* function allows one address to approve another address to spend tokens on his behalf. The standard implementation of the *approve* function is shown in Fig 32. We assume that *Alice* and *Tom* are two Ethereum users, and *Tom* runs an Ethereum node. The following steps reveal how *Tom* uses the *transaction order dependence* bug to monetize:
  - *Step 1*: Assume *Alice* has approved *Tom* to spend n of the tokens she holds. Now *Alice* decides to change *Tom*'s quota to m tokens, so *Alice* sends a transaction to modify *Tom*'s quota.

- *Step 2*: Since *Tom* runs an Ethereum node, he knows that *Alice* will change his quota to m tokens. Then, *Tom* sends a transaction (e.g., Using the *transferFrom* function to transfer n tokens to himself) to spend *Alice*'s n tokens, and pays a lot of *gas* to make his transaction executed first.
- *Step 3*: The node that obtains the accounting right packs transactions. Because *Tom* pays more *gas*, *Tom*'s transaction will be executed before *Alice*'s transaction. Therefore, *Tom* spent n tokens of *Alice* first and then is granted a quota of m tokens by *Alice*, which caused *Alice* to suffer losses.

```
4     function approve(address spender, uint256 value)
5     public returns (bool) {
6         require(spender != address(0));
7         _allowed[msg.sender][spender] = value;
8         emit Approval(msg.sender, spender, value);
9         return true;
10    }
```

Figure 32: The standard implementation of the *approve* function in the *ERC20* token standard.

- *Detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The contract contains the *approve* function in the *ERC20* token standard.
  - In the *approve* function, the quota of the approved address is set from one nonzero value to another nonzero value.

31. *Storage Overlap Attack*:

- *Cause*: All data in the smart contract share common storage space. If the data is arbitrarily written into the storage, it may cause the data to overwrite each other.
- *Consequence*: This kind of bugs will cause state variables to be overwritten, which may affect the function of the contract
- *Example*: Considering the contract in Fig 33, the attacker can call the *PushBonusCode* function arbitrarily, thereby writing data to the contract's storage, which may cause the *owner* variable to be overwritten.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - When modifying storage variables or writing new values to storage, the contract does not perform identity verification.

32. *byte[]*:

- *Cause*: The *byte[]* type can act as a byte array, but due to padding rules, it wastes 31 *bytes* of space for each element. It is better to use the *bytes* type instead.

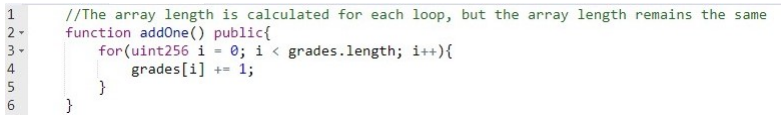
```
pragma solidity 0.6.2;
contract Wallet {
    uint[] private bonusCodes;
    address private owner;
    constructor() public {
        bonusCodes = new uint[](0);
        owner = msg.sender;
    }
    fallback () external payable {
    }
    function PushBonusCode(uint c) external {
        bonusCodes.push(c);
    }
    function PopBonusCode() external {
        require(0 <= bonusCodes.length);
        bonusCodes.pop();
    }
    function UpdateBonusCodeAt(uint idx, uint c) external {
        require(idx < bonusCodes.length);
        bonusCodes[idx] = c;
    }
    function Destroy() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }
}
```

Figure 33: A function contains a *storage overlap attack* bug

- *Consequence*: In the case of achieving the same function, this kind of bugs will cause more *gas* consumption.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Use *byte[]* type variables in the contract.

33. *Invariants in Loop*:

- *Cause*: Placing the invariant in the loop causes extra *gas* consumption.
- *Consequence*: The consequences of this bug are the same as the *byte[]* bug.
- *Example*: Consider the function in Fig 34, *grades.length* has not changed during the execution of the loop, but each execution of the loop will calculate *grades.length*, which consumes more *gas*. The recommended practice is to use a local variable to save the value of *grades.length*, and then use this local variable in the loop.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - There are some invariants in the loop (including the condition judgment part).

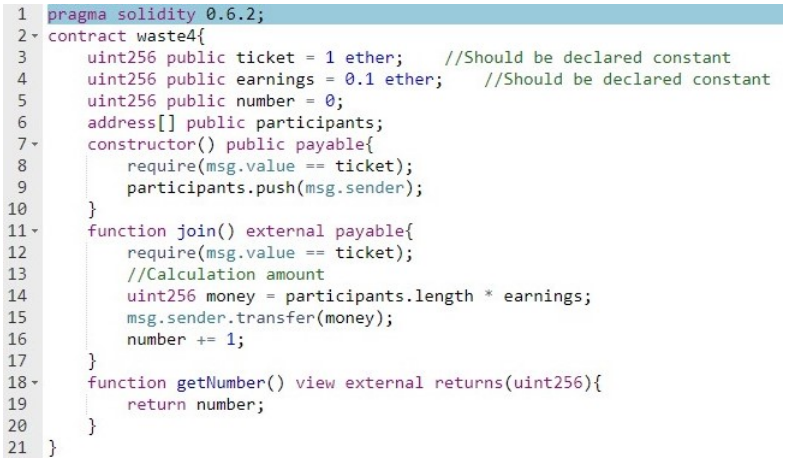


```
1    //The array length is calculated for each loop, but the array length remains the same
2    function addOne() public{
3        for(uint256 i = 0; i < grades.length; i++){
4            grades[i] += 1;
5        }
6    }
```

Figure 34: A function contains a *invariants in loop* bug

34. *Invariant State Variables Are Not Declared Constant*:

    - *Cause*: The contract declares invariants but does not use the *constant* keyword to modify the invariants, which will cause more *gas* to be consumed.
    - *Consequence*: The consequences of this bug are the same as the *byte[]* bug.
    - *Example*: Consider the contract in Fig 35. The operations in the contract will not change the value of *ticket* (line 3) and *earnings* (line 4), but these variables are not modified with the *constant* keyword, which leads to increased *gas* consumption.



```
1  pragma solidity 0.6.2;
2  contract waste4{
3      uint256 public ticket = 1 ether;    //Should be declared constant
4      uint256 public earnings = 0.1 ether;    //Should be declared constant
5      uint256 public number = 0;
6      address[] public participants;
7      constructor() public payable{
8          require(msg.value == ticket);
9          participants.push(msg.sender);
10     }
11     function join() external payable{
12         require(msg.value == ticket);
13         //Calculation amount
14         uint256 money = participants.length * earnings;
15         msg.sender.transfer(money);
16         number += 1;
17     }
18     function getNumber() view external returns(uint256){
19         return number;
20     }
21 }
```

Figure 35: A function contains two *invariant state variables are not declared constant* bugs

    - *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
        - There are constants in the state variable that will not be modified.
        - But these constants are not modified with the *constant* keyword.

35. *Unused Public Functions Within The Contracts Should Be Declared External*:

- *Cause*: Deploying a function with *public* visibility consumes more *gas* than deploying a function with *external* visibility. If a *public* function is not used in the contract, then declaring the function as *external* can reduce gas consumption.
- *Consequence*: The consequences of this bug are the same as the *byte[]* bug.
- *Example*: Considering the contract in Fig 36, the visibility of the function *sayHappyBirthday* and *refund* are both designated as *public* but are not called by other functions in the contract, which leads to an increase in the amount of *gas* consumed for deploying and calling the contract.

```
pragma solidity 0.6.2;
contract waste3{
    uint256 public contractBirthday;
    address public owner;
    constructor() public payable{
        require(msg.value >= 2 ether);
        contractBirthday = now;
        owner = msg.sender;
    }
    function sayHappyBirthday() public{
        if(now <= contractBirthday + 1 weeks)
            msg.sender.transfer(0.1 ether);
    }
    function refund() public{
        require(msg.sender == owner);
        if(now > contractBirthday + 1 weeks)
            selfdestruct(msg.sender);
    }
}
```

Figure 36: A contract contains two *unused public functions within the contracts should be declared external* bugs

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - There is the *public* function in the contract that are not called by other functions in the contract.

36. *Replay Attack*:

- *Cause*: Since Ethereum's public chain has been forked many times, Ethereum now has many chains. Therefore, if the verification value used by a transaction can be predicted, an attacker can replay the transaction on another chain.
- *Consequence*: This kind of bugs may cause the attacker to replay the transaction on another chain, causing economic losses to the contract owner.

- *Example*: Consider the function in Fig 37, where signatures are used for verification (line 10, 11), but signatures that have already been verified are not stored, which allows the user who invoked the contract this time to replay the transaction in another chain.

```
1   function transfer(
2         bytes memory _signature,
3         address _to,
4         uint256 _value,
5         uint256 _gasPrice,
6         uint256 _nonce)
7       public
8     returns (bool)
9     {
10      bytes32 txid = keccak256(abi.encodePacked(getTransferHash(_to, _value, _gasPrice, _nonce), _signature));
11      require(!signatureUsed[txid]);
12      address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);
13      require(balances[from] > _value);
14      balances[from] -= _value;
15      balances[_to] += _value;
16      signatureUsed[txid] = true;
17      return true;
18    }
```

Figure 37: A function contains a *replay attack* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The signature is used for verification in the contract.
  - But the contract does not store the signature value that has been processed.

37. *Suicide Contracts*:

- *Cause*: Authority control must be performed before a self-destructing operation, otherwise, the contract can be easily killed by an attacker.
- *Consequence*: This kind of bugs will cause the contract to be killed by anyone, and it will affect the function of the contract and cause economic losses.
- *Example*: Consider the contract in Fig 38. Because the *suicideSelf* function lacks authority control, any user can call the *suicideSelf* function to kill the contract and transfer the ethers in the contract to the address he specified.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There is at least one *selfdestruct-statement/suicide-statement* (without comments) in the contract.
  - The lack of authority control on the *selfdestruct-statement/suicide-statement* in the contract.

38. *Use tx.origin For Authentication*:

```
pragma solidity 0.6.2;
contract SuicideEasily{
    address public owner;
    constructor() public payable{
        owner = msg.sender;
        require(msg.value > 0);
    }
    modifier OnlyOwner{
        require(msg.sender == owner);
        _;
    }
    function withdraw() external OnlyOwner{
        msg.sender.transfer(address(this).balance);
    }
    function suicideSelf(address payable _Beneficiary) external{
        selfdestruct(_Beneficiary);
    }
}
```

Figure 38: A function contains a *suicide contracts* bug

- *Cause*: *Solidity* provides the keyword *tx.origin* to indicate the initiator of the transaction. Do not use *tx.origin* for authentication. Because when an attacker deceives your trust, the attacker can trick you into sending a transaction to a malicious contract deployed by the attacker, and then the malicious contract forwards the transaction to your contract. At this point, the originator of the transaction is you, so the attacker can be authenticated.
- *Consequence*: This kind of bugs will allow the attacker to pass the identity verification, which may affect the function of the contract or cause economic losses.
- *Example*: Consider the contract in Fig 39. Suppose the attacker deploys a contract *attack*, which includes a function *forward*. The function of *forward* is to call the *withdraw* function in contract *NobadTxorigin*. At this time, the attacker can induce the owner of the contract *NobadTxorigin* to call the *attack.forward* function to pass the authentication of the *withdraw* function (this call's *tx.origin* is the owner of the contract *NobadTxorigin*), and then send the ethers in the contract *NobadTxorigin* to the attacker.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The contract uses *tx.origin* for authentication (eg., *tx.origin == owner*).
  - But this statement (*tx.origin == msg.sender*) does not belong to the case of *use tx.origin for authentication*.

39. *Wasteful Contracts*:

- *Cause*: A contract that anyone can withdraw the ethers is called a *wasteful contract*, and the reason for this bug is that the contract does not have authority control over the withdraw ethers.

```
pragma solidity 0.6.2;
contract NobadTxorigin{
    address public owner;
    constructor() public payable{
        owner = msg.sender;
        require(msg.value > 0);
    }
    function withdraw() external{
        require(tx.origin == owner);
        msg.sender.transfer(address(this).balance);
    }
}
```

Figure 39: A contract contains a *use tx.origin for authentication* bug

- *Consequence*: This kind of bugs will cause any user to take ethers from the contract, which will cause economic losses.
- *Example*: Consider the contract in Fig 40. The contract *prodigal* defines the function modifier *onlyOwner*, but does not use the modifier to modify the function *withdraw*, which causes any user to take ethers from the contract.

```
pragma solidity 0.6.2;
contract prodigal{
    address public owner;
    constructor() public payable{
        owner = msg.sender;
        require(msg.value > 0);
    }
    modifier onlyOnwer{
        require(msg.sender == owner);
        _;
    }
    function withdraw() external{
        msg.sender.transfer(address(this).balance);
    }
}
```

Figure 40: A contract contains a *wasteful contracts* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - Users can remove Ether from the contract.
  - The contract does not perform identity verification or balance checking on the operation of taking ethers from the contract.

40. *Wrong Constructor Name*:

- *Cause*: *Solidity* allows developers to use a function with the same name as the contract as a constructor. If the developers misspell the name of the constructor, it will make the constructor a public function that anyone can call (up to and including version 0.4.26).

- *Consequence*: This kind of bugs causes the *constructor* to become a *public* function that can be called by any user, which may result in state variables being modified or economically lost.
- *Example*: Considering the contract in Fig 41, the developes mistype the constructor name (*MissContructor -> missContructor*), so that any user to call the function *missContructor* to change the *owner* variable, resulting in economic losses.

```
1  pragma solidity 0.4.26;
2  contract MissContructor{
3      address public owner;
4      function missContructor() public payable{
5          owner = msg.sender;
6          require(msg.value > 0);
7      }
8      function getMyBalance() external{
9          require(msg.sender == owner);
10         msg.sender.transfer(address(this).balance);
11     }
12 }
```

Figure 41: A contract contains a *wrong constructor name* bug

- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - There is no function declared as a constructor (using the *constructor* keyword or the same name as the contract) in the contract.
  - But there is a function in the contract whose name is similar to the contract name.

41. *Non-public Variables Are Accessed By Public/external Functions*:

- *Cause*: *Solidity* needs to specify the visibility of state variables, of which *internal* and *private* specify that state variables can only be accessed internally. However, using *public* or *external* functions to access *internal* and *private* state variables does not result in compilation errors.
- *Consequence*: This kind of bugs causes users to access externally invisible state variables by calling *public/external* functions, which will affect the function of the contract.
- *Example*: Considering the contract in Fig 42, the *owner* variables with *private* visibility can be accessed by functions with *external* visibility (*resetOwner*, *getOwner*), which causes other contracts/users to call these functions to access the *owner* variables.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:

```
1  pragma solidity 0.6.2;
2  contract privatePass {
3    uint private owner;
4    constructor(uint i_owner) public {
5      owner = i_owner;
6    }
7    function resetOwner() external {
8      owner = 0;
9    }
10   function getOwner() view external returns(uint256){
11     return owner;
12   }
13 }
```

Figure 42: A contract contains a *non-public variables are accessed by public/external functions* bug

  - State variables whose visibility is *private* or *internal* in the contract can be accessed by functions declared as *public* or *external*.

42. *Public Data*:

    - *Cause*: For miners, all contract codes and the values of the state variables are visible, even if visibility is specified using *private* or *internal*.
    - *Consequence*: This kind of bugs will cause miners to obtain the privacy/secret of the contract, which may affect the function of the contract or cause economic loss.
    - *Example*: Considering the contract in Fig 43, the value of the variable (*password*) declared in the 4th line of code is visible to the miner, so the miner can call the *replacePassword* function to modify the user's password, or call the *withdraw* function to take out the ethers in the contract.
    - *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
      - There are state variables with *private* visibility in the contract.

43. *Implicit Visibility Level*:

    - *Cause*: Failure to explicitly specify visibility can make the code difficult to understand (up to and including version 0.4.26. After version 0.4.26, functions must manually specify visibility, but state variables can still not specify visibility).
    - *Consequence*: This kind of bugs will reduce the readability and maintainability of the contract source code. When the default visibility of *Solidity*'s functions/state variables changes in the future, contract semantics will be difficult to understand.

```
1  pragma solidity 0.6.2;
2  contract OpenWallet {
3      struct Wallet {
4          bytes32 password;
5          uint balance;
6      }
7      mapping(uint => Wallet) private wallets;    //For miners, the password is visible
8      function createAnAccount(uint256 _account, bytes32 _password) public{
9          wallets[_account].balance = 0;
10         wallets[_account].password = _password;
11     }
12     function replacePassword(uint _wallet, bytes32 _previous, bytes32 _new) public {
13         require(_previous == wallets[_wallet].password);
14         wallets[_wallet].password = _new;
15     }
16     function deposit(uint _wallet) public payable {
17         wallets[_wallet].balance += msg.value;
18     }
19     function withdraw(uint _wallet, bytes32 _password, uint _value) public {
20         require(wallets[_wallet].password == _password);
21         require(wallets[_wallet].balance >= _value);
22         msg.sender.transfer(_value);
23     }
24 }
```

Figure 43: A contract contains the *public data* bug

- *Example*: Consider the contract in Fig 44. None of the state variables and functions in the contract explicitly specify visibility. This leads to developers having no way to quickly and accurately understand the external visibility of state variables and functions when reviewing code.
- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - There are state variables or functions in the contract that do not explicitly indicate visibility.

44. *Nonstandard Naming*:

- *Cause*: *Solidity* specifies a standard naming scheme. Following the standard naming scheme will make the source code easier to understand.
- *Consequence*: This kind of bugs will reduce the readability and maintainability of the source code. When developers review the code, they cannot quickly and accurately understand the type of each identifier.
- *Example*: Consider the contract in Fig 45. According to the naming convention, the constants declared in the code on line 3 should use uppercase letters and use _ to connect the words, and the function declared on line 7 should start with a lowercase letter. When the contract size is large (the number of lines of code is large), these incorrect naming will make it difficult for developers to quickly and accurately understand the type of each identifier.

```
pragma solidity 0.4.26;
contract NoVisitLevel {
    uint storeduint1 = 15;
    uint constant constuint = 16;
    uint32 investmentsDeadlineTimeStamp = uint32(now);
    bytes16 string1 = "test1";
    bytes32 string2 = "test1236";
    string  string3 = "lets string something";
    mapping (address => uint)  uints1;
    mapping (address => DeviceData) structs1;
    uint[] uintarray;
    DeviceData[] deviceDataArray;
    struct DeviceData {
        string deviceBrand;
        string deviceYear;
        string batteryWearLevel;
    }
    constructor() public{
    }
    function getConstuint() pure returns(uint256){
        return constuint;
    }
}
```

Figure 44: A contract contains a *implicit visibility level* bug

```
pragma solidity 0.6.2;
contract WrongName{
    uint256 public constant betprice = 0.1 ether;   //should be uppercase and use _ to connect two words
    address[] public users;
    event bet(address indexed _user);   //the 1st char should be uppercase
    //the 1st char should be lowercase
    function Bet() external payable{
        require(msg.value == betprice);
        users.push(msg.sender);
        emit bet(msg.sender);
    }
}
```

Figure 45: A contract contains a *nonstandard naming* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - The naming of the identifiers does not comply with the naming conventions specified in the *Solidity official document*[1].

45. *Too Many Digits*:

- *Cause*: Writing many consecutive numbers makes the code difficult to read and review. Developers can use scientific notation or exponential notation as an alternative.
- *Consequence*: This kind of bugs will reduce the readability and maintainability of the source code, and may also lead to assigning incorrect values to variables. If the value associated with ethers (tokens) is wrong, it will cause economic losses.

[1] `https://solidity.readthedocs.io/en/v0.6.8/style-guide.html#naming-conventions`

- *Example*: Consider the contract in Fig 46. In the *pay* function, the developers original intention is to receive 1 ether (*onceEther*) each time. However, because the variable *onceEther* was assigned the wrong value (0.1 ether) in the 3rd line of code, the ethers received by the *pay* function each time dropped to 0.1 ether.

```
1  pragma solidity 0.6.2;
2  contract tooManyDigits{
3      uint256 public onceEther = 1000000000000000000; //actual 10^19
4      address[] public users;
5      address payable owner;
6      constructor() public{
7          owner = msg.sender;
8      }
9      function withdraw() external{
10         require(msg.sender == owner);
11         owner.transfer(address(this).balance);
12     }
13     function pay() external payable{
14         require(msg.value == onceEther);    //1 ether 1 time
15         users.push(msg.sender);
16     }
17 }
```

Figure 46: A contract contains a *too many digits* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - In the source code (without comments), multiple digits appear consecutively.

46. *Unlimited Compiler Versions*:

- *Cause*: In different versions of *Solidity*, the same statement may have different semantics. When writing contracts, the *Solidity* version should be explicitly specified.
- *Consequence*: This kind of bugs leads to a decrease in the maintainability of the contract. In future versions of *Solidity*, the same sentence may have different semantics. Therefore, not specifying the specific compiler versions will result in the inability to correctly understand the original meaning of the code when reviewing the code in the future.
- *detect criteria*: This kind of bugs will exist when the contract has the following characteristics at the same time:
  - The *version-pragma-statement* contains the $\wedge$ symbol.
  - Or the *version-pragma-statement* contains the $>=$ symbol but not the $<$ symbol.

47. *Use Deprecated Built-in Symbols*:

- *Cause*: After *Solidity* version 0.5.0, several built-in symbols were discarded and replaced with other alternative built-in symbols (up to and including version 0.4.26).
- *Consequence*: This kind of bugs will reduce the code quality.
- *Example*: The contract in Fig 47 uses several deprecated built-in symbols. Try to replace them with their replacements. For a more complete deprecated built-in symbols and their corresponding replacements, please view this URL[2].

```
pragma solidity 0.4.26;
contract Old{
    address owner;
   bytes32 bytesnum;
   bytes32 hashnum;
    //function that has same name with contract's
    //is deprecated
    function Old(uint256 num){
        owner = msg.sender;
        //sha3 is deprecated
       bytesnum =sha3(num);
        //var is deprecated
        var a = 0;
        //block.blockhash is deprecated
        hashnum = block.blockhash(num); }
    //constant is deprecated
    function getOwner() constant returns(address){
        return owner; }
    function callAnother(){
        //callcode and throw are deprecated
        if(owner.callcode("")){ throw; }
        else{
            //suicide is deprecated
            require(msg.sender == owner);
            suicide(owner);}}
    function () external{
        //msg.gas is deprecated
        if(msg.gas > 2300){
            throw;}}
}
```

Figure 47: A contract contains a *use deprecated built-in symbols* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - Developers use *Solidity* before version 0.5.0 to write smart contracts and use built-in symbols that have been deprecated.

48. *View/constant Function Changes Contract State*:

- *Cause*: The keywords *view* and *constant* are provided in *Solidity* to modify functions, which means that these functions only read data

[2] `https://swcregistry.io/docs/SWC-111`

from the blockchain without modifying the data. However, such rules are not mandatory, so developers can modify data in functions declared as *view* or *constant* (up to and including version 0.4.26).

- *Consequence*: This kind of bugs may cause the abnormal function of the external contract calling this contract.
- *Example*: Consider the contract in Fig 48, although both the function *get* and the function *getPlusTwo* have been modified with the *view* or *constant* keyword, then both of these functions can change the contract state (line 8, line 12) without compiling error (just the compilation warnings).

```
1  pragma solidity 0.4.26;
2  contract Constant{
3      uint counter;
4      constructor(uint256 _counter) public{
5          counter = _counter;
6      }
7      function get() public view returns(uint){
8          counter = counter +1;
9          return counter;
10     }
11     function getPlusTwo() external constant returns(uint){
12         counter += 2;
13         return counter;
14     }
15 }
```

Figure 48: A contract contains a *view/constant function changes contract state* bug

- *detect criteria*: This kind of bugs exists when the contract contains the following characteristics:
  - There are functions declared as *view* or *constant* in the contract, and there are statements in these functions that can modify the state of the contract.

49. *Improper Use Of Require, Assert, And Revert*:

- *Cause*: *Solidity* provides several statements (*require*, *assert*, and *revert*) to handle errors. Although these statements are error handling statements, they are slightly different when used, so they need to be used correctly.
- *Consequence*: This kind of bugs may cause the abnormal function of the contract, and in serious cases, the function of the contract will be completely invalid.
- *Example*: Consider the contract in Fig 49. Due to the improper use of the *assert-statement* (line 9, the conditional judgment part always returns false), the *check* function is completely invalid.

```
1  pragma solidity 0.6.2;
2  contract GasModel{
3      uint x = 100;
4      function check() external{
5          uint a = gasleft();
6          x = x + 1;
7          uint b = gasleft();
8          //always false
9          assert(b > a);
10     }
11 }
```

Figure 49: A contract contains the *improper use of require, assert, and revert* bug

- *detect criteria*: This kind of bugs exists when the contract meets any of the following characteristics:
  (a) The *require-statement*, *assert-statement* or *revert-statement* causes the contract to be DOS under any circumstances.
  (b) Use the *require-statement* to handle internal errors or check invariants.
  (c) Use the *assert-statement* to check the external input or return value.