Detecting Vulnerabilities in Smart Contracts Based on Multidimensional Analysis Model

LI Jingwei[1]　ZHANG Han[2]　JIANG Shiyu[1]
XIE Ruihua[1]　WEN Weiping *
(1. School of Software & Microelectronics, Peking University, Beijing 100871;
2. Dingyuan Lanjian Information Technology Co. , Ltd, Changsha 410205.)

Abstract: With the development of blockchain technology, the application of cryptocurrency has become more extensive. Ethereum, as one of the most representative cryptocurrencies, provides a freer blockchain technology-smart contract. The new procedural trading mode created by smart contract has been widely used in sensitive services such as transaction payment and login authentication, and has good scalability. But at the same time, it also brings a new attack surface and more security threats. In order to solve the security problem of smart contracts, this paper proposes an automated method for detecting smart contract vulnerabilities. The syntax tree of the smart contract is used to generate the multidimensional analysis model consisting of control flow graph, the function call flow graph and the data dependency graph, and the vulnerability generation principle is combined to realize the vulnerability detection of the smart contract. In the analysis of 15394 contract addresses from Ethereum, the method proposed in the paper found that there was more than one security vulnerability in 4772 contract addresses, and the average detection vulnerability is about 80%. At the same time, compared with other vulnerability detection methods, it has better detection results.

Key Words: Smart Contract; Vulnerability Detection; Data Flow Analysis

Since the first introduction of Bitcoin in 2009, the endless stream of digital currencies has received increasing attention and is considered to be one of the most groundbreaking technologies in recent years. Built on cryptographic-based secure communications and distributed computing, digital currency is a decen-

作者简介:李经纬(1995—),男(汉),籍贯(辽宁省),硕士生
通信作者:文伟平,博士,教授,weipingwen@ss. pku. edu. cn

tralized monetary system based on blockchain technology. Blockchain can be thought of as a public verifiable data structure that provides data security for digital currency transactions. The data structure is implemented on a point-to-point network, and the nodes of the entire network must follow a consensus mechanism to generate and update data, which is responsible for managing the transaction processing and maintaining the consistency of the blockchain.

Bitcoin is currently the most popular blockchain application since the birth of digital currency, with a peak market capitalization of \$30 billion. As of January 2019, there were 2086 digital currencies in the world. Bitcoin accounted for 51.9% of the total market value of digital currency in the world. Ethereum and Ripco were the runner-up and runner-up respectively[1]. In 2013, the Ethereum[2], which was ranked second in the market share, was created by programmer Vitalik Buterin. Buterin has made distributed computing not limited to digital currency transactions, and designed the Ethereum blockchain. Ethereum has largely expanded the way in which consensus mechanisms deal with transactions. It allows trading operations to execute program code that is turing-complete, which we call smart contracts. Ethereum also introduced concepts such as virtual machine and smart contracts into blockchain platform for the first time. It is also the most active blockchain platform created and used by smart contracts.

Smart contracts are contracts that have the ability to execute code, allowing for business models such as equity-based crowdfunding, patent authentication and end-to-end credit, or shared wallets that need to be approved by multiple owners before executing a transaction. In Ethereum, the basic building language of smart contracts is a high-level programming language called Solidity, which implements the functions of smart contracts through Solidity language and compiles into binary bytecodes suitable for Ethereum virtual machines. Participants in the contract can interact with the deployed smart contracts through Ethereum trading operations and use a consensus mechanism to ensure that the program is executed correctly in the EVM.

Of course, the complexity of Ethereum brings more security risks. From The DAO security incident[3] to BEC's Batch Overflow vulnerability, smart contract vulnerabilities directly or indirectly lead to economic losses of hundreds of millions of dollars. Vulnerability detection research on smart contract vulnerabilities is becoming more and more necessary. In order to effectively detect smart contract vulnerabilities, we have made the following contributions in this paper:

1)Effective vulnerability Detection. Using the syntax tree of the smart contract to generate the multidimensional analysis model consisting of control flow graph, function call flow graph and data dependency graph, combined with the vulnerability feature, an effective vulnerability detection is performed on the smart contract.

2) Vulnerability Feature Library Construction. By analyzing a large number of smart contract vulnerability codes, the vulnerability features of 11 kinds of smart contract vulnerabilities are analyzed, and the smart contract vulnerability feature library is formed.

3) Experimental Evaluation. Through a large number of tests, the proposed smart contract vulnerability detection method is evaluated. The experimental results show that the proposed detection method has good accuracy and detection efficiency, and the average detection vulnerability is about 80%.

1 Methodology

Since most static audit methods[4-6] do not execute applications, by analyzing the function call relationship, data flow and other information, it is detected whether there is abnormal behavior inside the function, which greatly speeds up the speed of security audit analysis. Based on static analysis and detection, this paper designs a smart contract vulnerability detection method for security audit analysis of Solidity code. It includes three steps as shown in Figure 1: pretreatment, contract flow graph analysis, and vulnerability feature analysis.

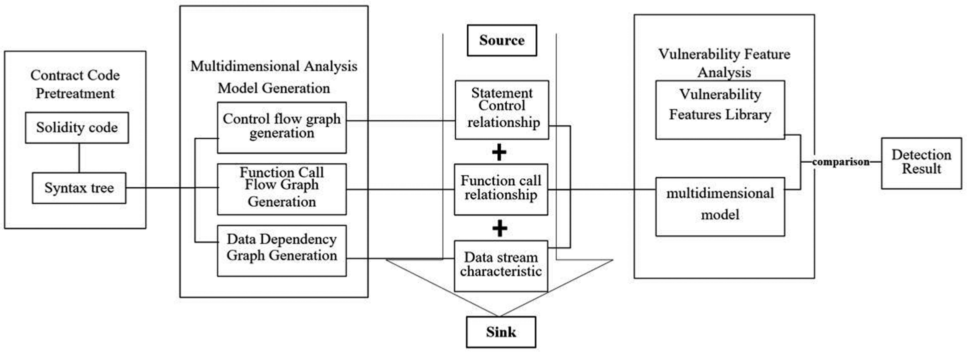
Contract Code Pretreatment
Solidity code
Syntax tree
Multidimensional Analysis Model Generation
Control flow graph generation
Function Call Flow Graph Generation
Data Dependency Graph Generation
Source
Statement Control relationship
+
Function call relationship
+
Data stream characteristic
Sink
Vulnerability Feature Analysis
Vulnerability Features Library
multidimensional model
comparison
Detection Result

Figure 1 Detecting Vulnerabilities in Smart Contracts Based on Multidimensional Analysis Model

1) Contract Code Pretreatment. The code feature is extracted from the detection code and parsed by the Solidity code to generate a syntax tree;

2)Multidimensional Analysis Model Generation. Based on the function, using the syntax tree to generate a control flow graph, get the statement control relationship of the contract code; Use the syntax tree as input to generate a function call flow graph, and observe the parameter transfer between functions, get the function call relationship in the contract code. The data dependency graph is further generated by the control flow graph to observe the data flow dependency between the basic blocks inside the function, get the data stream characteristics;

3) Vulnerability Feature Analysis. Base on the control flow graph, function call flow graph and data dependency graph, we get statement control relationships, function call relationships, and data flow features, then combine them into a multidimensional model, and compare the model with the vulnerability signature database to get the vulnerability analysis result of the contract.

1.1 Contract Code Pretreatment

During the pretreatment process, the input smart contract code is parsed by the contract syntax tree. The grammar tree analysis uses the recursive descent analysis method, and the analysis includes statement node analysis, expression node analysis and keyword node analysis. The specific statement nodes of the analysis are shown in Table. 1, and the "：" in the table is an assignment symbol, such as "a：e＋1", which means that the expression "e＋1" is represented by the symbol a. The symbol "|" means "or". The symbol "?" indicates that the previous part appears 0 to n times. For example, the production "a?" means that a appears 0 to n times, and n is any positive integer. Finally, when the last terminator match is completed, the contract syntax tree is parsed and an abstract syntax tree is generated.

Table. 1 Syntax Tree Node Production Specification Table

Syntax tree node	Node grammar production	Explain
contractDefinition	contract|library @Id{… }	Create a contract node or library contract node
eventDefinition	function|event @Id(v?){S}	Create an event node
functionDefinition	function @Id(v?){S}	Create function declaration node
block	{S}	Create a code block node

续表

Syntax tree node	Node grammar production	Explain
variableDeclara-tionStatement	v|e;	Create a variable declaration node
variableDeclara-tion	v:T @Id (= e)? ;?	Create a variable declaration or assignment node, denoted by v
typeName	T	Create a type node, denoted by T
returnStatement	return e	Create a return statement node
statement	S : if Statement; while Statement; for Statement; block; continue Statement; break Statement; return Statement; throw Statement; variable Declaration State-ment;	Create a statement node, denoted by S. Including if statement node, while loop statement node, for loop statement node, code block node, continue statement node, break statement node, return statement node, throw statement node, variable de-claration statement node.
expressionState-ment	e	Create an expression node, denoted by e
identifier	@Id	Create a keyword node, represented by @Id
ifStatement	if e S (else S)?;	Createa if statement node
functionCall	e ((T @Id)?);	Create a function call node
whileStatement	while (e) S;	Create a while loop statement node
forStatement	for(S;e;e)S;	Create a for loop statement node
structDefintion	struct @Id{v?};	Createa struct structure node
throwStatement	throw;	Createa throw statement node
breakStatement	break;	Create abreak statement node
continueStatment	continue;	Create acontinue statement node

The contract code converted into the form of an abstract syntax tree. The logic and data structure of the smart contrat was expressed in the form of a syntax tree, which can improve the analysis efficiency of the contract code while omitting the syntax details. This paper analyzes the different symbols in the contract based on the recursive descent analysis algorithm, where each non-terminal symbol corresponds to a parsing process. The program parses from the beginning of the contract code until it resolves to the last terminator.

1. 2 Multidimensional Analysis Model Generation

1. 2. 1 Control flow graph generation

Control flow graph is mostly used for intermediate abstract representations of program analysis. We use control flow graph here to represent the control flow of the contract for subsequent analysis. We define the Contract Control Flow Graph (CFG) as a directed graph that can be represented as a binary group CFG=(N,E). Where N is the set of nodes of the control flow and E is the set of tedge elements of the control flow. We store the control flow graph CFG using the storage method of the cross-linked list, and represent the nodes in the node set N with the following data structure. Basic Block is the basic block of the node. The contract source code is represented by the abstract syntax tree, including the start and end lines of the code. Firstin represents a collection of edges pointing to the current node, and firstout represents a collection of edges with the current node as the starting node.

```
class CFGNode {
    BasicBlock block = null;
    CFGEdge firstin = null;
    CFGEdge firstout = null;
}
```

The control flow edge in the edge set E is represented by the following data structure. verNum is the starting point of the current edge, and headVex is the end point of the current edge. HeadLink points to theneighbor that ends at the same point as the current edge. AdjLink points to the neighbor that starts at the same point as the current edge.

```
class CFGEdge {
    int verNum =-1;
    int headVex =-1;
    CFGEdge headLink = null;
    CFGedge adjLink = null;
}
```

The control flow graph mainly analyzes four types of statements in the Solidity language grammar, including sequential statements, conditional branch statements, jump statements, and loop statements. As shown in Table. 2, the sequential statement refers to the assignment operation and the comparison operation.

Table 2 Comparison Table of Statement Types and Keywords in Solidity Grammar

StatementType	Keywords
Sequential Statements	Arithmetic assignment operation, comparison operation: +,-,*,/,>,<,=,!,~,&,|,<<,>>,^,?
Conditional Branch Statements	if, require, assert
Jump Statements	Function Call, return, continue, break, throw
Loop Statements	while, for, do

Definition: A basic code block is a collection of code that contains multiple instructions. The program will run from the beginning of the collection to the end of the collection. The process of creating a control flow graph is the process of breaking down the program code into individual basic blocks and edges.

When we create a basic block, we follow the following rules:

1) Create a basic block based on the first statement of the current function;

2) When entering a conditional branch statement, jump statement, or loop statement, the current statement is used as the termination statement of the basic block of the current analysis and a new basic block is created;

3) Add the sequence statement to the current basic block.

The following is an algorithm description of the control flow graph we created:

```
INPUT:  Abstract Syntax Tree(AST)
OUTPUT: Control Flow Graph(CFG), Function Call Stack(FCS)
INITIALIZATION:
    Initialize the start node Start, end node End and empty node CFGNode of the generated graph;
    Assign CFGNode to the child node of start, and then use CFGNode as the basic block of the current analysis;
    set the basic block stack and the jump stack to be empty;

BEGIN:
for each ASTNode in AST do:
    switch(ASTNode):
      ifASTNode is Sequential Statement:
        Add the current statement to the current block;
        Break;
      end if

      if ASTNode is Conditional Branch Statement:
        Put the condition into the current block and end the current block;
        pushing the basic block onto the basic block stack;
        Recursively call the algorithm that creates the CFG to create the else part of control flow graph, and use the head node of the created graph as the child of the current block;
        Break;
      end if

      if ASTNode is Jump Statement:
        Save the statement to the current block and end the current block;
        Pushing the basic block onto the basic block stack;
        Find the target address of the jump statement in basic block stack;

        if found:
          Point the child pointer of the current block to the found basic block;
        end if

        if not found:
          Save the current block pointer and the number of lines of the jump statement into the jump stack;
```

```
            end if

            if the statement is a Function Call Statement:
                The current function definition AST, the called function, and the incoming
        parameters are logged to the function call stack FuncCallStack;
                Create an empty node as the current block;
                    Break;
            end if
        end if

        if ASTNode is Loop Statement:
            Put the condition into the current block, and end the current block, pushing the bas-
        ic block into the basic block stack;
            Recursively call the algorithm that creates CFG, and use the node that executes the
        internal flow graph as the child of the current block;
            The End of the node of the created graph points to the previous basic block;
            break;
            end if
    end swich

    Re-traverse the jump stack, add jump edges;
    Re-traverse the control flow graph, find the basic block whose control pointer is empty
in the flow graph, and point its child pointer to the End node;
end for
```

1.2.2 Function Call Flow Graph Generation

The generated control flow graph mainly analyzes the control flow within the subroutine, but also pay attention to the parameter transfer and dependencies between functions in the contract code audit. Because the trigger point of the vulnerability and the actual flawed code are often not in a function area. The analysis of the function call flow requires the declaration of the function as a collection of nodes and the function call as a collection of edges of the node. The following is an algorithm description for creating a function call flow:

```
INPUT: Abstract syntax tree(AST), function call stack(FCS)
    OUTPUT: function call flow graph(FCG)
    INITIALIZATION:
        If the process is the first call, apply the start node Start, end node End, and
empty node FCGNode of the generated graph, set the successor node of start to FCGNode,
and set FCGNode as the current node;

    BEGIN:
    for each ASTNode in AST do:
        if ASTNode isFunction definition grammar:
          Add function node;
          break;
        end if

        for each funcCall in FCS do:
          Add a pointer from the calling function to the called function;
          break;
        end for

        Re-traversing the function call flow graph, find the basic block whose child
pointer is empty in the function call flow graph, and point its child pointer to the End
node;

    end for
```

1.2.3 Data Dependency Graph Generation

In the program code, the input data is passed through the conversion of different statements. Therefore, it is necessary to analyze the operational behavior between programs and the dependencies between data variables. A data dependency graph is a directed graph that records data dependencies between nodes. For the sake of easy understanding, we define that the variable v_2 data depends on the variable v_1. If and only for the storage space of the variable v_2, there is an execution path: the variable v_1 is a write operation to the storage space, and the variable v_2 is a read operation to the storage space. The following is an algorithm description of data dependency analysis:

```
INPUT: Control Flow Graph(CFG)
    OUTPUT: Data Dependency Graph(DDG)
    INITIALIZATION 1:
        If the process is the first call, apply for the start node Start, end node End, and
empty node DDGNode of the generated graph, set the successor node of start to DDGNode,
and set DDGNode as the current node;
    INITIALIZATION 2:
        Add all global variables to the Globals list if the procedure is the first call

    BEGIN:
    for each CFGEdge in (All sides in the control flow graph) do:
        if CFGEdge isCFGNode:
            for each sentences in Nodes do:
                if sentences isVariable declaration:
                    Add the current variable to the current node and create a new node while re-
cording the CFGNode it belongs to;
                    break;
                end if

                //$v_{n-1}\cdots v_1$ represents all variables to the right of the equation
                //op represents the operator of all data, such as: +, -, *, /, >, <, =, !,
~, &, |, <<, >>, ^, ?
                if sentences is Assignment statement $v_n = v_{n-1}$ op$\cdots$ $v_1$:
                    If the data node list contains the variable $v_{n-1}$, retrieve $v_{n-1}$ in Globals;
                    If $v_n$ is not included in the data node list, create a new node $v_n$ and record
the CFGNode to which it belongs;
                    Add a side that points to $v_n$ by $v_{n-1}$;
                    Traverse all variables until variable $v_1$;
                    break;
                end if
                break;
            end for
        end if

        Re-traversing the data dependency graph, finding the basic block in which the
child pointer in the data dependency graph is empty, and pointing its child pointer to the
end node;

    end for
```

So far, based on the control flow chart, function call flow chart, and data dependency graph, we create a contract analysis multidimensional model containing statement control relationships, function call relationships, and data flow features, which including structure information, control flow information, variable transfer information, function call information and data dependency information.

1.3 Vulnerability Feature Analysis Module

The analysis of vulnerability features requires a comparison of the pre-constructed vulnerability model with the contract multidimensional model. The feature model of the vulnerability is mainly the feature function for each vulnerability and the data flow model necessary to trigger the vulnerability.

As shown in Table. 3. , the Source is defined as an untrusted input source, the Bug is a vulnerability feature, and the Sink is a sensitive operation point. all the parameters controllable by the attacker in the smart contract are regarded as Source, which mainly includes the parameters of all functions, all global variables of the contract, and msg. data, msg. sender, tx. origin, msg. sender. value. Transfer operations, modify amounts, execute code, the operations that can modify any global variables and selfdestruct, suicide, call, callcode, delegatecall, transfer, send as sensitive operations Sink.

Defines that the contract is in a vulnerable state if and only if there is a control flow from Source to Sink, while the control flow contains corresponding features that match the vulnerability features that are expected to be defined.

Table 3 Corresponding Operations of Source and Sink Nodes

Smart contract analysis node	Node feature
Untrusted Input Source	The parameters of all functions, all global variables of the contract, and msg. data, msg. sender, tx. origin, msg. sender. value
Sensitive Operating Point Sink	Transfer operations, modify amounts, execute code, the operations that can modify any global variables andselfdestruct, suicide, call, callcode, delegatecall, transfer, send

Take the reentry vulnerability as an example to illustrate how to construct a vulnerability model in advance. Reentry vulnerabilities are also called recursive call vulnerabilities. In the "TheDAO" security incident[3], attackers used reentry vulnerabilities to steal more than 360,000 Ethereum, resulting in more than \$60 million in economic losses.

```
01:  contract MyWallet{
02:     mapping(asress => unint256) public balances;
03:     function depositFunds() public payable{
04:        balance[msg. sender] += msg. value;
05:     }
06:     function withdrawFunds (unint256 _money) public{
07:        require (balances[msg. sender] >= _money);
08:        // limit the withdrawal
09:        require(msg. sender. call. value(_money)());
10:     }
11:  }
```

This is a wallet contract code with a reentrant vulnerability. Its function is to recharge and withdraw cash. The contract does not take into account the possibility that the called external user account is a contract account. After the external contract account calls the withdrawal function of the wallet contract, the withdrawal function runs to the 7th line code, and the wallet contract is transferred to the external contract account, thereby triggering the callback function of the external contract. Once the external contract calls the withdrawal function of the wallet contract again, because the intermediate data of the user asset has not been updated, the second transfer operation can still be triggered. An attacker can use this vulnerability to transfer all of the Ethereum assets owned by the contract. Therefore, the reentry vulnerability is to repeatedly call the attack contract recursively, and then recursively call the vulnerability function of the vulnerability contract through the attack contract.

By analyzing the syntax tree of the reentrant vulnerability, we find that the trigger of the vulnerability is that the function can be repeatedly entered, and the intermediate variables that the conditional statement depends on are not changed in time. In the analysis and detection process, Source is defined as an untrusted input source, Reentrancy Bug is defined as the vulnerability trigger point, and Sink is the operation of modifying the contract book. According to the analysis, there are three conditions for the reentry vulnerability to exist: 1)

There is a transfer operation; 2) The transfer operation is before the modification of the contract book; 3)There is no limit to the number of transfers. The vulnerability feature fields “call”, “callcode”, and “degegatecall” are thus obtained. Since the number of transfers will not be limited if the transfer transaction is initiated in the above several ways.

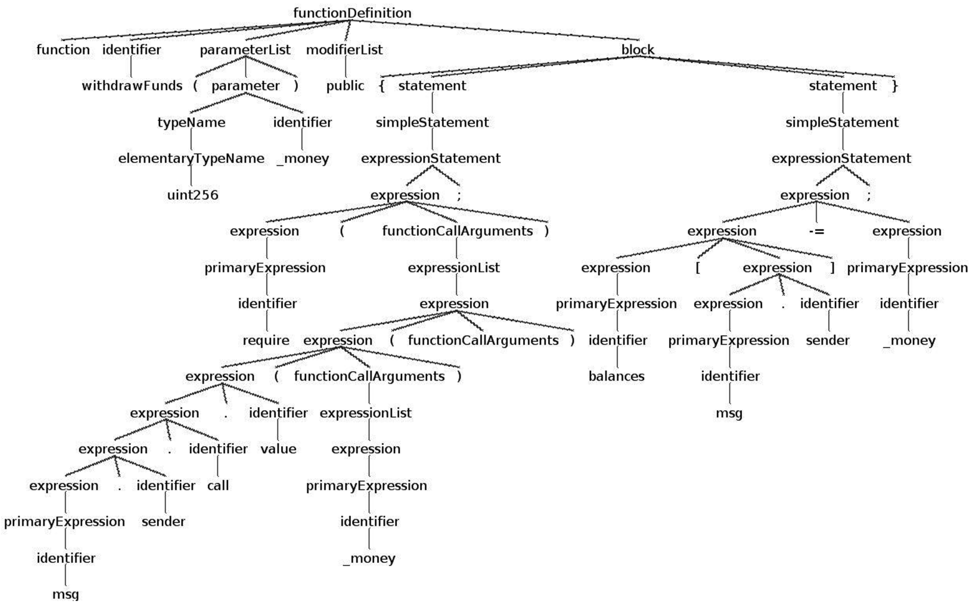
functionDefinition
function identifier parameterList modifierList block
withdrawFunds (parameter) public { statement statement }
typeName identifier simpleStatement simpleStatement
elementaryTypeName _money expressionStatement expressionStatement
uint256 expression ; expression ;
expression (functionCallArguments) expression -= expression
primaryExpression expressionList expression [expression] primaryExpression
identifier expression primaryExpression expression . identifier identifier
require expression (functionCallArguments) identifier primaryExpression sender _money
expression (functionCallArguments) balances identifier
expression . identifier expressionList msg
expression . identifier value expression
expression . identifier call primaryExpression
primaryExpression sender identifier
identifier _money
msg

Figure 2　Schematic Diagram of the Minimal Syntax Tree of the Vulnerability Contract

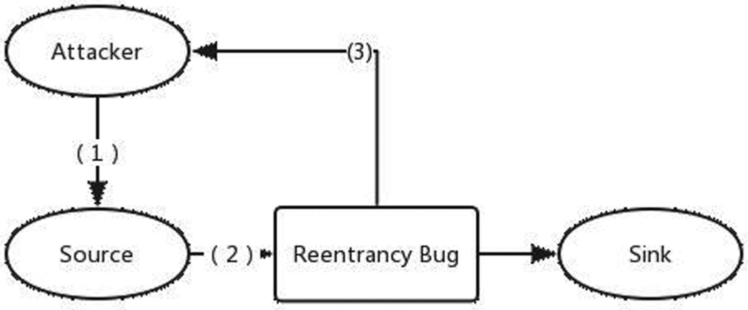

Figure 3　Schematic Diagram of the Reentry Data Flow Model

As shown in Table 4, by analyzing a large number of smart contract codes, we summarized the features of 11 kinds of vulnerabilities such as reentrant vulnerability.

Table 4 Smart Contract Vulnerability Features Library

Vulnerability type	**Vulnerability feature**
Reentrant Vulnerability	call,callcode,degegatecall
Numeric Overflow Vulnerability	+,−,*,/,int,uint
Time Dependent Vulnerability	now,block. timestamp
Random Number Generation Vulnerability	now, block. timestamp, tx. gasprice, msg. gas, block. number, block. hash, sha256, sha3, ripemd160,ecrecover
Self-destructive Vulnerability	suicide, selfdestruct
PermissionCcontrol Vulnerability	tx. orgin,owner
Return Value Judgment Vulnerability	Send and other functions whose return value is boolean
Code Execution Vulnerability	delegatecall
Denial of Service Vulnerability	transfer,send
Variable Coverage Vulnerability	Same variable name, function name

2 Evaluation

In order to fully test the performance of the proposed smart contractvulnerability detection method, we chose two different data sets. The first data set is the most active 15394 contract addresses in Ethereum as of April 2019. Ethereum Scan is able to perform a search and verification of the verified contract. This paper analyzes the contract code of the publicly available 15394 contract addresses. These contracts are all still in operation and have been analyzed to include a total of 75,799 contracts.

We design the following experiments to verify the results and performance: 1) Verify the test results on the Ethereum contract; 2) Compare this test method with other test tools; 3) Evaluate the non-functional test results of the method. The test environment is shown in Table. 5.

Table 5　Experimental Environment

CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Memory	4096 MB
Docker	Docker version 18.06.1-ce, build e68fc7a
Ubuntu	Linux ubuntu 4.4.0-31-generic ＃50～14.04.1-Ubuntu
Python	Python 3.6.1
Java	java version “1.8.0_131”
Php	5.4.45
Solc	4.0.25
Geth	1.8.17-stable-8bbe7207
Truffle	Truffle v4.1.14
Remix	1.01
Node-js	v0.10.25

2.1　Detection Performance

In the evaluation process of vulnerability detection，several standards for evaluating the performance of vulnerability detection methods are mainly referred,which include Accuracy(ACC)，false positive rate (FPR). The specific contents of these indicators are as follows：

Accuracy (ACC)：the percentage of the number of vulnerabilities correctly detected by the vulnerability detection method in the true number of vulnerabilities in thesmart contract source code.

Falsepositive rate (FPR)：the percentage of vulnerabilities detected by vulnerability detection methods that do not actually exist in the program，that is，the percentage of the number of system error reports in the total number of reports；

The test results of the method are verified by the combination of manual analysis，and the results are shown in Table 6. On average，one out of every 10 contracts has a vulnerability. Of the 15394 contract addresses，4772 contract addresses have vulnerabilities，accounting for approximately 31％.

Table 6　Test Results of Ethereum Contract Data Set

category	**Number of vulnerabilities**	**Number of contracts with vulnerabilities**	**The proportion of the vulnerability contracts in the total number of contracts tested**	**ACC**
Reentry	10	9	0.06%	83.3%
Numerical overflow	2983	2427	15.85%	76.3%
Time-dependent	103	103	0.67%	88%
Self-destruction	23	23	0.15%	70.4%
Permission Control	76	67	0.44%	73.6%
Return Value Judgment	5961	4839	31.59%	75%
Code Execution	21	18	0.12%	70%
Random Number Generation	189	178	1.16%	60%
Conditional Competition	256	232	1.51%	78%
Denial of Service	3	3	0.02%	100%
Variable Coverage	339	315	2.06%	97%

In this section, the vulnerability detection accuracy rate of this method is analyzed by manual verification contract. Among them, the key code of a detected numerical overflow vulnerability is as follows. The code comes from the digital token RocketCoin①. RocketCoin was created in 2017, and the token value during the peak period was $12.44.

```
function multiTransfer(address[] _addresses, uint[] _amounts) public returns (bool success) {
    require(addresses.length <= 100
                    && _addresses.length== _amounts.length);
    uint totalAmount;
    for (uint a = 0; a < _amounts.length; a++){
      totalAmount += _amounts[a];
    }
    require(totalAmount > 0 && balances[msg.sender] >= totalAmount);
    balances[msg.sender]-= totalAmount;
    for (uint b = 0; b < _addresses.length; b++) {
      if (_amounts[b] > 0) {
        balances[_addresses[b]] += _amounts[b];
        Transfer(msg.sender, _addresses[b], _amounts[b]);
      }
    }
```

① https://etherscan.io/token/0x6fc9c554c2363805673f18b3a2b1912cce8bfb8a

In this function, totalAmount refers to the total transfer amount, and _amounts represents an array of amounts for each recipient's separate transfer. The method proposed in this paper detects a numerical overflow vulnerability in line 5 of the code. The contract code does not check that the sum of the total transfer amount and the current transfer amount must be greater than the value of either one. During the manual verification process, we created a virtual Ethereum node and re-registered two test account addresses.

The test account address created by the node:

1)"0x2360114bef2b0bb8d5dc3a506599f19506c76ff5",

2)"0x36f2c9bdb7b15727145d35d6f9ef233514c0277d"。

After the contract with the vulnerability is successfully deployed in the virtual node using the first account, the second account is used to launch the attack in the previous way, and the second account successfully adds a large number of digital tokens. The vulnerability is verified successfully.

2.2 Detection of CVE Vulnerabilities

To further illustrate the detection capabilities of this method, we detect vulnerability code that has become a security event and vulnerability code that has a CVE number. We test more than ten vulnerability codes with CVE numbers. Table 7 lists the results of some CVE tests, demonstrating the ability of this method to detect contract vulnerabilities.

Table 7 Test Results of CVE Vulnerabilities

Token name	Vulnerability type	CVE number	Public time	Results
DAOToken	Reentry	—	2016.06	Successful
BEC[7]	Numerical Overflow	CVE-2018-10299	2018.04	Successful
BTCR[8]	Numerical Overflow	CVE-2018-11687	2018.05	Successful
PKT[9]	Numerical Overflow	CVE-2018-11809	2018.06	Successful
RMC[10]	Numerical Overflow	CVE-2018-12230	2018.06	Successful
CryptoFlower[11]	Numerical Overflow	CVE-2018-13525	2018.07	Successful

2.3 Comparison withRelated Approaches

The data set used in the comparison test is the Ethereine real contract data set in the previous section, comparing the averageaccuracy rate and false positive rate of several tools, include Oyente[12], Mythril[13] and Tencent Smart Contract Security Detection System.

As shown in Figure. 4, the comparison test results show that our method has higher accuracy and lowest false positive rate than other tools. In general, the detection of symbolic execution will result in higher accuracy and a higher false positive rate. The figure clearly shows that Oyente and Mythril have higher false positive rates. Oyente's accuracy is not high, due to its small number of vulnerability detection types. TencentDetect has a high rate of false positives due to lax restrictions. In summary, the false positive rate and accuracy of our method have reached a good balance, and it is more suitable as an audit assistant for security personnel analysis.

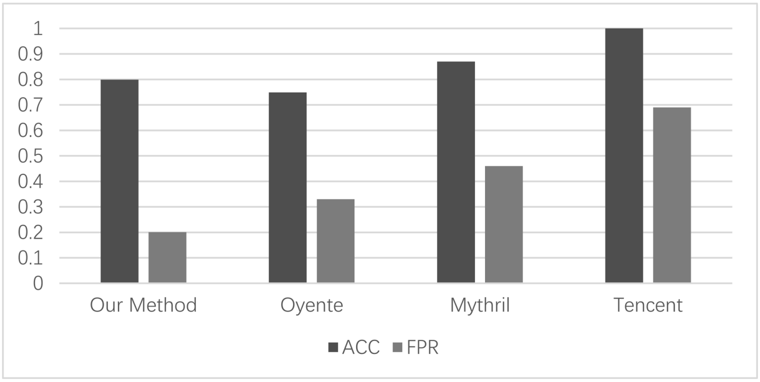

Figure 4　Comparison Test Results

2. 4　Run-time Performance

In addition to the functional detection of vulnerability detection methods, we also focus on non-functional test results. Usually, this test includes the calculation test of time and space resources. The test of space resources is also divided into memory consumption and hard disk space consumption. However, in the problem of detecting smart contract vulnerability, the loss of hard disk space is negligible, and the consumption of memory resources is obviously different due to the implementation method. So here we mainly test the difference in time. When testing the Ethereum real contract dataset, our method took an average of 29. 4 seconds to analyze a contract address, and 77 contracts (0. 5%) timed out or could not be resolved. 10, 545 contracts were safe, and about

68.5% of the contracts did not detect vulnerabilities. Oyente and Mythril require higher time and higher memory and CPU usage in the same test environment. However, TencentDetect's detection speed is the best, and a contract address can be detected in less than 10 seconds. This experiment also proves that our algorithms and detection methods have low time complexity and space complexity, and can complete contract vulnerability detection relatively quickly.

3 Related Work

As a product of the new era, smart contracts will surely become the trend of future development. It not only provides a variety of financial services, but also brings advantages in legal certification. However, while promoting social progress, it also faces unprecedented security challenges. Security researchers are also making unremitting efforts to build a secure and equitable blockchain ecosystem.

Delmolino[14] disclosed a variety of logical vulnerabilities in a smart contract built by themselves, including non-return of contract transfers, secret disclosures, and incentive failure. These security issues are caused by logical vulnerabilities in the design and implementation of contract.

Hirai[15] proposed a formal authentication for smart contract based on Isabelle high-order logic interaction theorem prover, which uses Lem language to define a formal model for EVM virtual machine, and proves the security features of EVM with existing interactive theorem. The main benefit of this framework is that they provide strong formal verification and are accurate and without false positives.

At the same time, in the field of smart contractvulnerability detection, several well-known tools have been proposed. Most of these tools use symbolic execution technology.

Luu[12] proposed Oyente, which can be used to detect reentry vulnerabilities or transaction order dependency vulnerabilities. However, the system also has some shortcomings. For example, it can only detect specific types of vulnerabilities, some of which can only be exploited by malicious miners.

In addition, Mythril[13] ,Maian[16] , SECURIFY[17] and other systems are also using this idea to carry out work. Maian also adopted a series of verification measures to reduce false positives.

Atzei[18] provided an analysis report on the type of attack on the Ethereum contract. The report assesses the smart contract attacks in recent years and classifies and summarizes the types of attacks.

In order to further analyze the security vulnerabilities of smart contracts, Matt Suiche[19] completed a decompilation tool for EVM bytecode.

Zhou[20] proposed an inverse tool Erays that can convert EVM binary bytecode into a high-level language pseudocode. Erays analyzes the information entropy and reuse of contract code and reduces the unknown portion of the undisclosed contract code by comparing it with published contracts.

Bhargavan[21] completed the initial work of converting Solidity and EVM bytecode into an existing formal verification system. Unfortunately, their articles are not analyzed in conjunction with real-world smart contracts.

Kalra[22] proposed a framework called ZEUS to detect security vulnerabilities and user-defined security policies for smart contracts. They compile smart contracts with user policies into LLVM-based intermediate code representations and then use existing LLVM-IR-based verification tools for further static analysis.

4 Conclusion

This paper uses the static analysis of the contract flowgraph to build a contract model that contains code structure information, control flow information, function call information and variable transfer information, and data dependency information. Data flow analysis of the contract model is compared with the vulnerability feature data flow model we have summarized to achieve the purpose of threat detection. Experiments show that the smart contract vulnerability detection method we designed can achieve a certain detection effect.

However, there are still deficiencies, and further improvements are needed in the following work: 1) Make the division of the vulnerability type clearer. In the summary work of the types of smart contract vulnerabilities, in addition to well-known vulnerabilities in several security incidents, there are no standardized rules for other vulnerabilities. There are 11 different types of smart contract vulnerabilities involved in this article. It is inevitable that some of the detection features of vulnerabilities have intersecting parts. In the future, the smart contract source code should be studied in more depth, and the type of vulnerability detection and model characteristics will be clearly defined; 2) The method designed in this paper can only detect the vulnerabilities of data streams that match the type of vulnerability. It is not good for the conditional vulnerability detection that is triggered by complex data vulnerabilities such as high concurrency; 3) This paper mainly uses static code analysis audit technology, which is not implemented in combination with dynamic code execution monitoring technology. In the future, we can consider further improving the audit system by adding dynamic monitoring technology to improve the accuracy of threat detection.

References

[1] Cryptocurrency Market. CoinMarketCap. [EB/OL]. [2019-01-27]. https://coinmarketcap. com.

[2] Buterin V. Ethereum: A next-generation smart contract and decentralized application platform[J]. URL https://github. com/ethereum/wiki/wiki/White-Paper 2014, 7.

[3] Phil Daian. Analysis of the DAO exploit. [EB/OL]. [2019-01-27]. http://hackingdistributed. com/2016/06/18/analysis-of-thedao-exploit.

[4] Ayewah N, Hovemeyer D, Morgenthaler J D, et al. Using static analysis to find bugs[J]. IEEE software, 2008, 25(5): 22-29.

[5] Baca D, Petersen K, Carlsson B, et al. Static code analysis to detect software security vulnerabilities-does experience matter? [C]//2009 International Conference on Availability, Reliability and Security. IEEE, 2009: 804-810.

[6] Zhang Lin,Zeng Qing-kai. Static Detecting Techniques of Software Security Flaws[J]. Computer Engineering, 2008, 34(12):157-159.

[7] Address of BeautyChain (BEC) contract. [EB/OL]. [2019-01-27]. https://etherscan. io/address/ 0x c5d105e63711398af9bbff092d4b6769c82f793d.

[8] Address of BTCR contract. [EB/OL]. [2019-01-27]. https://etherscan. io/token/0x6aac8cb9861 e42bf8259f5abdc6ae3ae89909e11.

[9] Address of PKT contract. [EB/OL]. [2019-01-27]. https://etherscan. io/token/0x2604fa406be 957e542beb89e6754fcde6815e83f.

[10] Address of RMC contract. [EB/OL]. [2019-01-27]. https://etherscan. io/token/0x7dc4f41294 697a7903c4027f6ac528c5d14cd7eb.

[11] Address of CryptoFlower contract. [EB/OL]. [2019-01-27]. https://

etherscan. io/token/ 0xe7b5b1c27ba88a4501ea81130dc97252ac5f3cbc.

[12] Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter [C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM(CSS'16), 2016: 254-269.

[13] ConsenSys Diligence, Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit. [EB/OL]. In HITBSecConf 2018. [2019-01-27]. https://github. com/ConsenSys/ mythril.

[14] Delmolino K, Arnett M, Kosba A, et al. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab[C]// International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2016: 79-94.

[15] Hirai Y. Formal verification of Deed contract in Ethereum name service [J]. November-2016. [Online]. Available: https://yoichihirai. com/ deed. pdf, 2016.

[16] Nikolicć I, Kolluri A, Sergey I, et al. Finding the greedy, prodigal, and suicidal contracts at scale[C]//Proceedings of the 34th Annual Computer Security Applications Conference. ACM, 2018: 653-663.

[17] Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: Practical security analysis of smart contracts[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018: 67-82.

[18] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok)[C]//International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2017: 164-186.

[19] Suiche M. Porosity: A decompiler for blockchain-based smart contracts byte-code[J]. DEF CON, 2017, 25.

[20] Zhou Y, Kumar D, Bakshi S, et al. Erays: reverse engineering ethereum's opaque smart contracts[C]//27th USENIX Security Symposium (USENIX18). 2018: 1371-1385.

[21] Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal verification of smart contracts: Short paper[C]//Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. ACM, 2016: 91-96.

[22] Kalra S, Goel S, Dhawan M, et al. Zeus: Analyzing safety of smart contracts[C]//25th Annual Network and Distributed System Security Symposium (NDSS18). 2018.