Received: 31 March 2019 | Revised: 29 June 2019 | Accepted: 25 July 2019

DOI: 10.1002/spe.2745

SPECIAL ISSUE PAPER

WILEY

NeuCheck: A more practical Ethereum smart contract security analysis tool

Ning Lu[1,2] | **Bin Wang**[1] | **Yongxin Zhang**[1] | **Wenbo Shi**[1] | **Christian Esposito**[3]

[1]College of Computer Science and Engineering, Northeastern University, Shenyang, China

[2]School of Computer Science and Technology, Xidian University, Xi'an, China

[3]Department of Computer Science, University of Salerno, Fisciano, Italy

Correspondence
Wenbo Shi, College of Computer Science and Engineering, Northeastern University, Shenyang, China.
Email: shiwb@neuq.edu.cn

Funding information
National Nature Science Foundation of China, Grant/Award Number: 61601107, U1708262 and 61472074; Fundamental Research Funds for the Central Universities, Grant/Award Number: N172304023; China Postdoctoral Science Foundation, Grant/Award Number: 2019M653568

Summary

Ethereum is one of the currently popular trading platform, where any one can exchange, buy, or sell cryptocurrencies. Smart contract, a computer program, can help Ethereum to encode rules or scripts for processing transactions. Because the smart contract usually handles large number of cryptocurrencies worth billions of dollars apiece, its security has gained considerable attention. In this paper, we first investigate the security of smart contracts running on the Ethereum and introduce several new security vulnerabilities that allow adversaries to exploit and gain financial benefits. Then, we propose a more practical smart contract analysis tool termed NeuCheck, in which we introduce the syntax tree in the syntactical analyzer to complete the transformation from source code to intermediate representation, and then adopt the open source library working with XML to analyze such tree. We have built a prototype of NeuCheck for Ethereum and evaluate it with over 52 000 existing Ethereum smart contracts. The results show that (1) our new documented vulnerabilities are prevalent; (2) NeuCheck improves the analysis speed by at least 17.2 times compared to other popular analysis tools (eg, Securify and Mythril; and (3) allows for cross-platform deployment.

KEYWORDS

blockchain, Ethereum, security analysis, smart contract

1 | INTRODUCTION

1.1 | Background and motivation

The last few years have witnessed the rise of decentralized cryptocurrency[1] as a digital medium of exchange devoted to secure financial transactions, control the creation of additional units, and verify the transfer of assets. Up to now, hundreds of cryptocurrencies have been introduced on the market, in which Ether is especially prevailing for its feature to encode rules and scripts for processing transactions. It is well known that Ethereum[2] is an Ether trading platform where users can trade, buy, and sell Ether, and this platform is built on blockchain, a decentralized and tamper-proof distributed ledger devoted to record cryptocurrency transactions. Smart contract, as a executable program running in such platform, can encode any set of transaction rules, such as paying deposit in escrow system and saving wallets. It is obvious that the security of smart contracts directly determines whether the transaction can work normally. Unfortunately, a tip-of-the-iceberg victim list has demonstrated the vulnerability of smart contract, such as Dice2win,[3] King of Ether,[4] and Accidental.[5] One latest notorious incident is DAO,[6] an Ethereum-based venture capital fund, in which hackers exploited its contract bugs to steal up to $60 million.[7] In addition, all participants, both vicious and benign, on the Ethereum network

can execute the deployed contract code with the current state of the blockchain, just because smart contract has its correct execution enforced by the consensus protocol. This means, once the smart contract has been deployed in the Ethereum, those vicious participants would analyze and attack the vulnerabilities of this contract. Thus, for the purpose of analyzing the vulnerabilities of smart contracts before deployment, it is necessary to design an effective smart contract security analysis tool.

This paper mainly focuses on the security analysis of Ethereum smart contracts. Considering that Ethereum, a permissionless blockchain platform possesses the characteristics of diversity, fast trading, and openness; an efficient smart contract security analysis tool should satisfy the following requirements: (1) wide range of vulnerability analysis. Since Ethereum would load lots of smart contracts in various application areas, the contract analysis tool should identify as many vulnerabilities as possible. (2) High analysis speed. Since the Ether price is always fluctuating with time and this makes the deployment of smart contracts have certain timeliness, the contract analysis tool should analyze the vulnerabilities as soon as possible. (3) Ability to deploy on cross-platform environment. Since Ethereum allows any users to deploy their smart contracts due to its openness and this makes users prefer to analyze the bugs in their own environment, the contract analysis tool should provide cross-platform deployment support. Above all, we will obtain a satisfactory smart contract analysis tool only by meeting these requirements.

1.2 | Limitation of prior art

A handful of smart contract security analysis tools have been proposed so far. Most of these tools are based on symbolic execution,[8] and their basic idea is to transform the smart contract source code or Ethereum virtual machine[9] byte code into an abstract interpretation through semantic analysis. Oyente, ZEUS, Mythril, Maian, and Securify are examples of classic smart contract security analysis tools. Oyente[10] is the first symbolic execution tool that can analyze Ethereum smart contracts to detect bugs. It can only be deployed on Windows and Mac systems and analyze a small number of smart contract vulnerabilities. ZEUS[11] adopts abstract interpretation and symbolic model to translate smart contract source code to the LLVM framework.[12] In order to reduce the semantic loss caused by the transformation process, ZEUS would generate numerous extra operation overhead. Mythril[13] uses symbolic analysis, taint analysis, and control flow checking to detect the security vulnerabilities. It uses Z3, a complex theorem proving tool, as the core detector, which would take large amount of time to complete the theorem proof. Hence, both ZEUS and Mythril are not practical due to the high time overhead. Maian[14] adopts interprocedural symbolic execution to transform the smart contract source code, and then uses concrete validator to filter the false positives reported by analysis tool. Securify[15] uses dependency graph to extract precise semantic information from the smart contracts, and then define semantic-based security patterns to verify compliance and violation. Owing to software compatibility, both Maian and Securify cannot run on Windows system. In summary, the existing analysis tools suffer the following disadvantages: fewer vulnerabilities that can be detected, lower analysis speed, and inability of deployment to across-platform environment.

1.3 | Our proposed smart contract security analysis tool

In this paper, we first document several new security vulnerabilities in the Ethereum smart contract and give examples of each vulnerability. They are as follows: access control vulnerability, the cause of which is that the developers of smart contracts ignore the definition of the access control modifier; reentrancy vulnerability, the cause of which is that some smart contract function sends transactions by call and this causes the function be called again("re-entered") before its previous invocation's complete execution; hash collision vulnerability, the cause of which is that the variables in smart contracts have the same hash value returned by the sha3 function; integer underflow vulnerability, the cause of which is that an arithmetic operation result in smart contracts exceeds the range of which can be represented with a given number of digits; and dependence on predictable variables vulnerability, the cause of which is that the smart contracts use predictable variables as random source to perform certain key operations.

More importantly, we design a more practical smart contract security analysis tool termed NeuCheck, which allows users to efficiently detect security vulnerabilities in the Ethereum smart contracts. Different from the semantic-based transformation in existing analysis tool,[10,11,13-15] we introduce the syntax tree in the syntactical analyzer to complete the transformation from source code to intermediate representation, so as to avoid missing semantic. NeuCheck first immediately parses the source code that cannot be modified after deployed into a syntax tree covering the whole contract; then, it adopts the open source library working with XML to analyze the syntax tree; finally, it explores the categories of the vulnerabilities and informs the users where the vulnerabilities are.

1.4 | Summary of experimental results

To measure the prevalence of our documented new vulnerabilities, at first, we collect more than 52 000 smart contracts from Ethereum since March 24, 2016 and run NeuCheck on this smart contract data set. The statistic results show that more than 90% of smart contracts potentially have the bugs. Just for our documented access control vulnerabilities, its number could reach as high as 79 842, and two or more vulnerabilities can exist in a single smart contract.

To prove the efficiency of our analytical tools, firstly, we measure accuracy by comparing NeuCheck with two well-known smart contract analysis tools, Securify,[15] and Mythril.[13] Secondly, we measure the time overhead of tools in analyzing different types of smart contracts by comparing our tool with the popular analysis tools Securify[15] and Mythril.[13] The experimental results show that NeuCheck improves the vulnerability analysis speed by at least 17.2 times.

1.5 | Key contributions

Our major contributions can be summarized as follows.

- We document several new smart contract bugs, including access control vulnerability, reentrancy vulnerability, hash collision vulnerability, integer overflow vulnerability, and dependence on predictable variable vulnerability (Section 4).
- We design a syntax tree parse-based smart contract security analysis tool termed as NeuCheck* (Section 5). Compared to the existing symbolic execution-based analysis tools, the benefits of NeuCheck are three-fold: (1) more vulnerabilities can be detected; (2) the analyzing speed is improved; and (3) the ability of deployment to dozens of cross-platform environments is supported.
- We run NeuCheck on the real Ethereum smart contract data set and evaluate its performance over different kinds of contracts (Section 6). The evaluation results demonstrate that (1) our new documented vulnerabilities are prevalent and (2) NeuCheck is more efficient.

The remainder of this paper is structured as follows. Background is presented in Section 2. Related work is presented in Section 3. The motivating examples are presented in Section 4. The evaluation of the analysis tool is presented in Section 5. The evaluation of the analysis tool is presented in Section 6. Finally, Section 7 concludes the paper.

2 | BACKGROUND

Blockchain, with decentralization, openness, and tamper-proof, is a new technology, which has received widespread attention. Blockchain and mobile edge computing[16] are considered the future of IoT,[17] eg, crowd-intelligence platform[18] is implemented using blockchain smart contract and mobile edge computing network, which together function as a trustless hybrid human-machine. In this section, we mainly introduce the basic principles and characteristics of both the Ethereum and smart contract. Based on this, we point out what requirements should be satisfied for an efficient smart contract security analysis tool.

2.1 | Ethereum

As a cryptocurrency, Ether is especially prevailing for its feature to encode rules and scripts for processing transactions. Ethereum is an Ether trading platform where users can trade, buy and sell Ether, and this platform is built on a peer-to-peer distributed ledger technology called blockchain. In Ethereum, each transaction would be packaged into the blocks in blockchain, and each block is cryptographically linked to the previous block in chronological order, thus making it irreversible. Its block structure is shown in Figure 1. The block header contains the following fields: the hash value of the previous block, the time stamp to represent the creation time of this block, the current difficulty to represent the blockchain workload, the Nonce to prove the workload, and the Merkle root generated by the hash value of all transactions in the block body. Through the consensus mechanism (eg, PoW,[19] PoS,[19] and PBFT[20]), all honest nodes maintain a consistent blockchain view. According to openness, the blockchain can be categorized into permission blockchain (eg, consortium blockchain, private blockchain) and permissionless blockchain (eg, public blockchain).

Generally speaking, Ethereum as permissionless blockchain platform in the cryptocurrency market has the following characteristics. (1) Openness. Theoretically, Ethereum allows anyone to participate the mining network without

*The source code has been uploaded to the free source code hosting platform called GitHub, and it is available at https://github.com/Northeastern-University-Blockchain/NeuCheck.

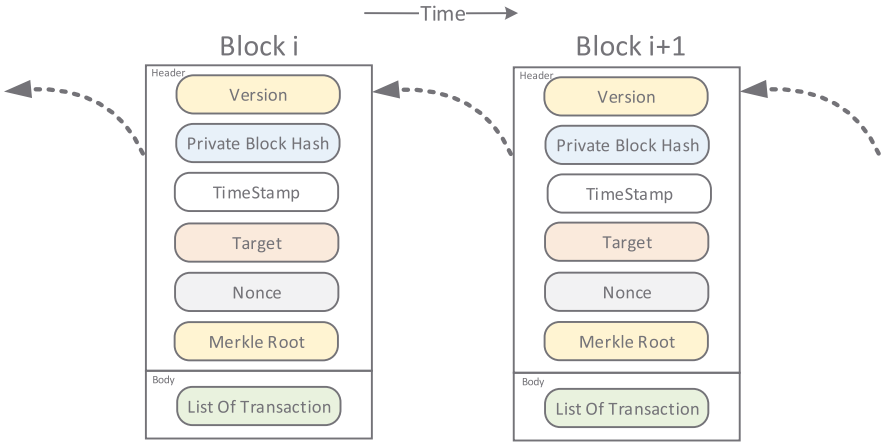

FIGURE 1 Blockchain structure diagram in Ethereum [Colour figure can be viewed at wileyonlinelibrary.com]

authorization, send transactions on the chain as their will, and participate in the process of forming consensus on the network at any time. (2) Fast trading. Ethereum handles millions of transactions one day, on average, amounting to hundreds of millions of dollars, and each transaction uses Ether, which is constantly fluctuating in value, as the settlement currency. (3) Application diversity. Ethereum can run the smart contracts, a Turing-complete script-language, and this could make it support for multiple applications. Since Ethereum has entered the Constantinople stage at now, it cannot only reduce the operating cost but also optimize the development experience of developers. To a certain extent, this situation will also promote the applicability of Ethereum even more.

2.2 | Smart contracts

A smart contract is a Turing-complete computer program that runs on the Ethereum platform, which is responsible for encoding any set of rules for processing transactions.[21] For instance, a contract can complete an electronic fund transfer when it is executed (eg, purchase virtual game currency from an escrow system). Recently, smart contract has seen steady adoption and could support hundreds of applications, including voting, crowdfunding, secret bidding, and multisignature wallets. In Ethereum, smart contract has the following features. (1) High value. Smart contracts can handle large number of cryptocurrencies worth millions dollars, and thus easily attract those greedy adversaries. (2) Permanent operation. Once a smart contract that contains serious security vulnerabilities has been deployed on the Ethereum, it cannot patch for bugs, no matter how much money it holds. (3) Data transparency. Since Ethereum adopts the consensus protocol to ensure that the smart contracts perform well, once a contract has been deployed, all participants on mining network would execute its code. Moreover, this can increase the risk of its security vulnerabilities being exposed. Above all, it is more urgent to analyze the security vulnerabilities of smart contract before deployment.

Reckoned with the features of Ethereum mentioned earlier, the Ethereum smart contract security analysis tool should satisfy the following requirements. (1) Wide analysis range. The more users adopt the smart contracts, the more vulnerabilities are exposed. In this, it should cover as many vulnerabilities as possible. (2) High analysis speed. The analysis speed would determine the deployment for the smart contract and further affect the duration of an entire transaction. In view of the volatility of cryptocurrencies, the smart contract should be analyzed, and the sooner the better. (3) Cross-platform deployment. As more and more users develop smart contracts for various business scenarios, they prefer analyzing the contracts on their own devices for convenience to uploading to third parties. Thus, this requires the analysis tool should be easily deployed on dozens of cross platform environments.

3 | RELATED WORK

As early as 2016, Kevin Delmolino et al[22] detected logical error and run-time error in smart contracts, such as call-stack bugs, blockhash, and Ethereum-specific incentive bugs. Then, Vitalik Buterin[23] listed and categorized the vulnerabilities in the Ethereum smart contracts. The subsequent series of smart contract security incidents also confirmed the importance of security for smart contracts. Our effort makes a great contribution to an efficient Ethereum smart contract security analysis tool, which is responsible for verifying the correctness and fairness of contracts before deployment.

To date, most of existing smart contract security analysis tools are based on symbolic execution. According to the interpretation method for vulnerabilities in smart contracts, they can be categorized as model formalization-oriented analysis tools and pattern formalization-oriented analysis tools.

Model formalization-oriented. Oyente[10] is the first symbolic execution based smart contract security analyzes tool, which can only be available on linux. Besides, it only detects timestamp dependence vulnerability and mishandled exception vulnerability, and its time overhead is large. With the help of Z3[24] (an SMT solver),[25] Mythril[13] uses symbolic analysis, taint analysis, and control flow checking to detect varieties of security vulnerabilities. Since Z3 requires proofing complicated theorem, its analysis time is long. In addition, its deployment requires many dependencies and this makes it hard to deploy. To sum up, the model formalization-oriented analysis tools would not only spend large amount of time but also detect fewer vulnerabilities.

Pattern formalization-oriented. Bhargavan et al[26] translated the Ethereum smart contract into F* framework[27] for analysis and verification, which enables the formal specification and verification of properties. However, it does not support many syntactic features of Solidity. Based on abstract interpretation and symbolic model, ZEUS[11] translates smart contracts to the LLVM framework, which needs some additional operations to keep semantics consistent with the original, making the analysis speed slow. As the first tool for precisely specifying and reasoning about trace properties, Maian[14] uses interprocedural symbolic analysis and a concrete validator to avoid false positives. Yet, it cannot run on Windows and its analysis speed is slow. To optimize the existing symbolic checker, Securify[15] uses a dependency graph to extract precise semantic information from the contracts, but the Soufflé used in it may not available on Windows. To sum up, the pattern formalization-oriented analysis tools provide strong formal verification to improve accuracy but are not practical due to the lower analysis speed and inability of deployment to across-platform environment.

Our work is fundamentally different with these early research. We first document several classes of new Ethereum smart contract bugs. Then, we propose a more practical smart contract security analysis tool, termed NeuCheck. We adopt a syntax tree in a symmetrical analyzer to complete the transformation from source code to intermediate representation, instead of symbolic execution, Then query on intermediate representation through dom4j to detect the vulnerabilities. Recently, Tikhomirov et al[28] proposed a new analysis tool, which is similar to NeuCheck. However, NeuCheck adopts a different methods of parsing IR and its analysis range is wider. Compared with previous work, NeuCheck has the following advantages: firstly, it can detect more vulnerabilities; secondly, improve the analysis speed; and thirdly, support the cross platform environment.

4 | NEWFOUND SECURITY VULNERABILITIES

In this section, we will motivate the problems we solve through actual security issues in smart contracts. We first document several new security vulnerabilities in the Ethereum smart contract that allows malicious users to exploit and gain financial benefits, and then give the relevant examples.

4.1 | Access control

Different functions in smart contracts should have different access rights. In most cases, normal functions would be set to the public type. Nevertheless, those core functions handling sensitive information should not be set to public type, or else access control vulnerabilities would occur.

```
contract initContract{
    function initContract() public{
    owner = msg.sender;
    }
}
```

Once the access permission of the contract constructor is set to public type, the attacker can easily modify the contract owner by calling the *initContract()* function. For example, the smart contract Parity[29] lost the Ether due to the incorrect setting on the access permission of the constructor. It was reported that, on Wednesday, July 19, 2017, an access control vulnerability in the multisignature wallet ("multi-sig") code used as part of Parity Wallet software was exploited by unknown parties. Three wallet accounts holding large balances of Ether were compromised, and the balances were moved into accounts held by the attacker.

```
contract Rubixi{
    address private owner;
    function DynamicPyramid(){
    owner = msg.sender;
     }
    function collectAllFees(){
    owner.send(collectedFees);
    }
    ...
}
```

Coincidentally, Rubixi[30] is a contract to implement a Ponzi scheme. The Ponzi scheme is a fraudulent, high-yield investment project in which participants benefit from newcomers' investments. Rubixi has changed the contract name during the development process. However, the programmer has forgotten to change the name of the constructor accordingly, and the original constructor becomes a function that anyone can call. As is shown in the code fragment, the *DynamicPyramid()* function sets the owner address, and the contract owner can withdraw the profit by calling the *collectAllFees()* function.

4.2 | Two kinds of reentrancy

In terms of the Ethereum transaction, both its atomicity and order guarantee that the called function cannot be reentered until the nonrecursive function terminates execution. Unfortunately, the fallback mechanism in Ethereum may allow an attacker to reenter the called function and further cause an unexpected situation. Reentry attacks can be divided into single function reentrancy and cross-functions reentrancy.

4.2.1 | Single function reentrancy

For example, we suppose that an attacker publishes an *Attack* contract, and the contract *Wallet* is already deployed on the blockchain.

```
contract Attack{
 function(){
 Wallet(msg.sender).transfer(this);
   }
}
```

```
contract Wallet{
 bool flag = false;
 function transfer(address destination){
 if (!flag){
 destination.call.value(2)();
 flag = true;
     }
   }
}
```

The *transfer()* function in the smart contract *Wallet* sends 2wei (the smallest unit of the Ether is wei, and 1 Ether = 10^{18} wei) to the parameter address *destination* by using the *call()* function. Obviously, the *call()* function has no gaslimit. We suppose that the attack contract *Attack* calls the *transfer()* function with its own address, the *call()* function calls the callback function of *Attack*, and the callback function calls the *transfer()* function again. We find that the variable flag has not been assigned "true", so the *if* condition is still true. In this case, the *transfer()* function would send 2wei to the parameter address *destination* again, and then the program starts the callback function and enters the loop call. Only if on the conditions that the program runs out of gas, the stack is full, or the account balance is insufficient can the loop stop. In addition, based on it, except the last transfer, all previous transfers would be true and valid. It is worth noting that the "DAO attack", which caused a large loss of Ether in June 2016, exploited this vulnerability.

4.2.2 | Cross-function reentrancy

An attacker can also perform similar attacks using two different functions that share the same state variables. In this case, the attacker first calls the *withdraw()* function and then calls the *transfer()* function in the callback function. Since the *balance* variable has not been set to 0, the attacker can still perform a transfer transaction even if the deposit returned by *withdraw()* has been received. There are some explanations of cross-function reentrancy attacks in the report.[31]

```
mapping (address => uint) private balances;
function transfer(address to, uint amount){
    if (balances[msg.sender] >= amount){
        balances[to] += amount;
        balances[msg.sender] -= amount;
    }
}
function withdraw() public{
    uint amountToWithdraw = balances[msg.sender];
    msg.sender.call.value(amountToWithdraw)();
    balances[msg.sender] = 0;
}
```

4.3 | Hash collision

It is well known that map is a data structure that implements an associative array abstract data type. A map use a hash function to compute an index into an array of slots, from which the desired value can be found. Generally speaking, in a map, the storage address of the data is actually the hash value of both the *map.key* and *map.slot*. This value is of type uint256, which can be expressed as *address(map_data) = sha3(key,slot)*. If the data is a structure, its members are sequentially stored in the storage, and the storage location is *sha3(key,slot) + offset*. That is, directly add the members' offset in the structure to the previously calculated hash value as the storage location. This *hash + offset* calculation of storage location directly causes the hash of the sha3 algorithm[32] to lose its meaning. Furthermore, in some cases, the method produces $sha3(key_1,slot) + offset = sha3(key_2,slot)$, which is a hash collision. For example, there are two maps in the contract *Project* and the structure *NameRecord* stored in the map *registeredNameRecord*. In such situations, a hash collision may occur.

```
contract Project
{
    struct NameRecord {
        bytes32 name;
        address mappedAddress;
    }
    mapping(address => NameRecord) public registeredNameRecord;
    mapping(bytes32 => address) public resolve;
    ...
}
```

Considering the case of an array, a fixed-length array of global variables in a smart contract is stored in the order of index. If it is a variable-length array, the storage location of the array member is selected according to the hash value, and its location is calculated in *sha3(address(array_object)) + index*. The length of the array is stored in the slot of the array. Variable-length arrays are also stored in the same way as *hash + offset*. In this case, the hash collision will occur, and this makes the value of *stateVar* cover the value of *balances*. In general, the hash returned by the sha3 method does not collide, but there is no guarantee that the $hash(mem_1) + n$ does not conflict with other $hash(mem_2)$.

```
contract Project
{
    mapping(address => uint) public balances;
    uint[] stateVar;
    ...
}
```

4.4 | Integer overflow

Once the integer variable has upper or lower bounds, the integer overflow would occur. Suppose that there is an out-of-bounds in the arithmetic operation. If the result is beyond the maximum representation range of the integer type, its value would be zero or a very small value. This situation often happens to traditional softwares because programmers are easy to make this mistake. To date, most of existing analysis tools (eg, SmartCheck[28]) can only detect integer overflow (eg, division overflow). Hence, this paper would complete such vulnerability.

```
contract overflow {
 function transfer(address[] rec, uint256 v)returns(bool){
 uint cnt = rec.length;
 uint256 amount = uint256(cnt) * v;
 require(balances[msg.sender] >= amount);
 ...
 for(uint i = 0;i < cnt;i++){
   balances[rec[i]] = balances[rec[i]].add(v);
   Transfer(msg.sender,rec[i],v);
   }
  return true;
  }
 }
```

This is an example from the real world.[25] This contract caused a loss of $28 billion due to an integer overflow. For 4th line statement *uint256 amount = uint256(cnt) * v* in *transfer()* function, it is an integer multiplication without any verification of the result obviously. In order to succeed in stealing massive amounts of tokens, the attacker can skillfully construct the parameter *v* and make the variable *amount=0* to avoid 5th line of verification.

```
contract Underflow{
  mapping(address => uint) balances;
  function Bank(address to,uint value)public returns(bool){
    if(balances[msg.sender] - value >= 0){
      balances[msg.sender] -= value;
      balances[to] += value;
    }
    return true;
  }
}
```

If uint is less than 0, it will cause an underflow, and the value will be assigned to the maximum value. In t*Bank()* function, the statement *balances[msg.sender] - value >= 0*, which avoids the check by integer underflow. In addition, more than 20 different scenarios have been introduced[33] to handle integer arithmetic.

The Solidity development team released the SafeMath library[34] on August 6, 2018. It contains secure mathematical calculations including common addition, subtraction, multiplication, and division operations but does not contain exponential operations. We recommend that developers of smart contracts prioritize the use of the SafeMath library for arithmetic operations.

4.5 | Dependence on predictable variables

Some smart contracts use predictable variables as triggers (eg, timestamp, coinbase, and gaslimit) to perform certain key operations (eg, sending Ether). The bug that may exist in these contract is called dependence on predictable variables vulnerability. So far, most of the tools (eg, Oyente[10] and SmartCheck[28]) document timestamp-dependent vulnerabilities and ignore coinbase, gaslimit, and number predictable variable vulnerabilities. Thus, this paper mainly supplements other such vulnerabilities.

```
contract theRun {
  uint private Last_Payout = 0;
  uint256 salt = block.timestamp;
  function random returns(uint256 result){
  uint256 y = salt * block.number/(salt%5);
  uint256 seed = block.number/3+(salt%300)+Last_Payout+y;
  uint256 h = uint256(block.blockhash(seed));
  return uint256(h%100) + 1;
  }
 }
```

Let us take the timestamp dependency as an example. A good example of a time-dependent contract is theRun contract, which uses a self-produced random number generator to determine who won the grand prize.[35] Technically, theRun uses the hash of one of the previous blocks as a random seed to select the winner. The block selection is determined by the current block timestamp. In the contract, the hash value of the previous block and the current block number is known, and other contract variables, such as *Last_Payout* to generate the random seeds, are also known. In this, the miner can

TABLE 1 Summary of the vulnerabilities

Vulnerabilities	Description
Reentrancy	(1) Using the call function as a transfer function; (2) no setting of the gaslimit; (3) status update after transfer.
Integer overflow and underflow	Improper use of arithmetic operators results in operations that exceed the maximum or minimum range of integer representations.
Hash collision	The data structure of a variable-length array or multimember map variable causes the hash value returned by the sha3 function to have the same value.
Access control	Ignoring defining the access control modifier of the contract constructor causes the core variable to be tampered with.
Unchecked call	The return value was not verified after calling the call function.
Dependence on predictable variables	Use variables such as timestamp, coinbase, gaslimit, and number as trigger conditions for performing certain key operations (for example, sending Ether).
DoS from external contracts	A conditional expression in a conditional statement should not depend on an external contract because the called contract may tamper with the return value affecting the execution result of the conditional statement.
Unchecked send	The return value was not verified after calling the send function.
Unknown libraries	Calling an unknown external library function may introduce an attack.
Unfixed compiler version	The specific version number of the Smart Contract Compiler is not specified.
tx.origin	Misuse of tx.origin indicates msg.sender, resulting in the transfer address not being the direct caller of the function but the original originator of the transaction.
Strict equality	An attacker can manipulate the contract logic by using strict equality on account balance.
Locked money	The smart contract can receive Ether but cannot send Ether.
Unsafe-type declaration	The variable type is not specified when the variable is declared; for example, the type of i in var i = 1; is the smallest integer type (uint 8).
Costly bytes	The gas consumption of byte [] is higher than for bytes, and we recommend using bytes instead of byte[].
Costly loops	Loops that contain many expensive computations may exceed the block gas limit. Ethereum is a very resource-constrained environment; loops with large or unknown numbers of steps should be avoided.
Unsafe inherit from token	A contract inherited from a contract with a name including the word "token," which may throw exceptions from inside one of the functions mentioned above.
Private modifier	In Solidity, the private modifier cannot hide variables, and anyone can access the code and data on the contract.
Unnecessary payable fallback function	Starting from Solidity 0.4.0, smart contracts do not need to reject unexpected payments manually. Therefore, the code function () payable{throw;} is unnecessary.
Unstandardized naming	In Solidity, function names usually start with a lowercase letter, whereas event names usually start with an uppercase letter.
Unspecified visibility level	Identify the visible levels of functions and variables to avoid confusion.

precalculate and select the timestamp, and this helps the function produce results that are beneficial to him or her. As a result, the opponent may completely bias the results of the random seed to any value, thereby rewarding the jackpot to any player he or she likes.

To analyze such vulnerabilities, we check for predictable state variables in variable declaration expressions and various assignment expressions. We treat contracts that meet these conditions as vulnerability contracts. The smart contract vulnerabilities detected by NeuCheck are shown in Table 1.

5 | THE NEUCHECK TOOL

We propose a smart contract analysis tool called NeuCheck to help (1) developers code safer smart contracts and (2) users identify smart contracts with security issues. It is worth mentioning that our tool employs modular management, which has good scalability. Because each vulnerability corresponds to a detection pattern, they do not interfere with each other.

5.1 | Design overview

In this paper, we design a security analysis tool to verify the correctness and fairness of Ethereum smart contracts before their deployment. It is well known that Solidity is the primary language to develop smart contracts on Ethereum platform at present. Although it has good readability, it is not structured data and thus unable to parse directly. Based on this, the

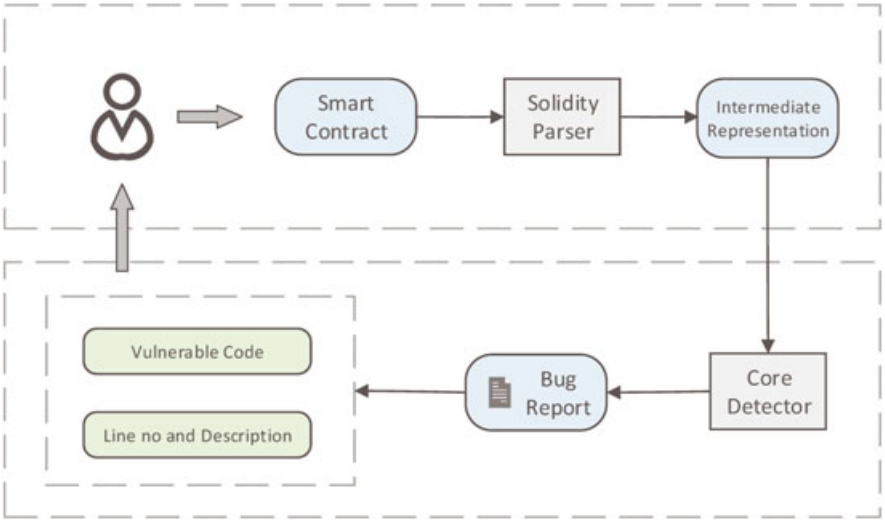

FIGURE 2 Overview architecture of NeuCheck [Colour figure can be viewed at wileyonlinelibrary.com]

management of the whole analysis process has been divided into three stages: (1) in order to format the structure of the smart contract, we need to convert its source code into the intermediate representation; (2) parse intermediate expression to verify its correctness and fairness; and (3) provide the security report to users including the type and location of the vulnerabilities, so as to help users locate potential security vulnerabilities.

There are two challenges during the security analysis process: **(1) How to complete intermediate representation transformation without semantic missing issues**. Because of semantic heterogeneity, the traditional semantic-based transformation would produce semantic missing. For this, we introduce the syntax tree in syntactical analyzer to complete the transformation from the smart contract source code to the intermediate representation. Different from the semantic-based method, we do not care about the grammatical meaning of the generated single token group and the relationship with the context during the analysis of smart contract data stream. Instead, we constitute a syntax tree in strict accordance with the syntax structure token group, and thus make the intermediate representation cover all source codes without any semantic missing. **(2) How to improve the analysis speed**. Symbolic execution-based analysis tools require logical solver to theoretically prove the reachability of the theorem during the vulnerability analysis, which increases the time overhead. Instead, We adopt dom4j, a more efficient open source library for working with XML to load the intermediate expression into memory in the form of "DOM tree", and then operate node objects through the API to complete detection. This method does not need logical solver, and thus saves the processing time.

Figure 2 shows the overall architecture of NeuCheck, which requires only smart contract source code as input to the system. NeuCheck outputs a security analysis report, showing the user the category and location of the vulnerabilities. NeuCheck is modular in design containing two main components: the Solidity parser and the Core detector. The prior parses the source code into a syntax tree covering the whole contract, and each statement in the contract is a branch of the syntax tree. The latter implements the detection patterns for the vulnerabilities discussed in Section 4. We represent the security vulnerabilities with expressions and then match the expressions to the syntax tree. If the match is successful, it means there is a vulnerability, and vice versa.

5.2 | Implementation

We implement NeuCheck in Java, which employs ANTLR,[36] a powerful parser generator, to complete intermediate representation transformation, and then uses dom4j to parse intermediate representation. Benefiting from the efficiency of dom4j and the compatibility of ANTLR, NeuCheck not only supports cross-platform deployment but also attains rapid analysis. In addition, NeuCheck covers a broader range of vulnerabilities, including the bugs mentioned in Section 4.

5.2.1 | Solidity parser

NeuCheck uses the Solidity parser built by ANTLR to transform the smart contract source code into an XML parse tree,[37] an intermediate representation. ANTLR is a powerful parser generator for reading, processing, or translating structured languages or binary files. It is widely used to build languages, tools, and frameworks because of its parsing capabilities keeping the flexibility and simplicity. The ANTLR parser could identify the valid input, regardless of its complexity, and construct a parse tree more easily traversable. We use its lexer to identify source code and convert it into discrete groups of

```
$ grun Solidity stateVariableDeclaration -tokens
uint token =1;
[@0,0:3='uint',<'uint'>,1:0]
[@1,5:9='token',<Identifier>,1:5]
[@2,11:11='=',<'='>,1:11]
[@3,13:13='1',<DecimalNumber>,1:13]
[@4,14:14=';',<';'>,1:14]
[@5,15:14=<EOF>,  EOF>,2:0]
```

FIGURE 3 Token stream

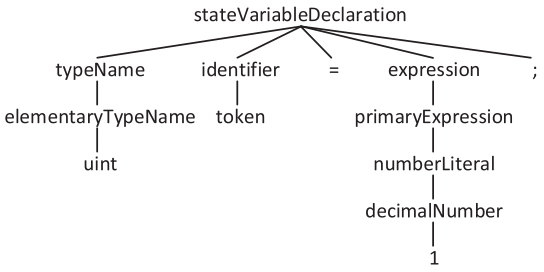

FIGURE 4 Parse tree

characters called tokens, including keywords, identifiers, symbols, and operators. Then, the parser organizes these tokens and transforms them into a valid sequence, such as a syntax tree, according to the given grammar.

We just use a piece of code uint token = 1; to explain the analysis process. There are two stages during the procedure. In the first stage, lexer would produce a token stream, as shown in Figure 3, when importing the code into ANTLR. Each output line represents a single token. For example, $[@0, 0 : 2 = 'uint', < 'uint' >, 1 : 0]$ indicates that the 'uint' is the first token at index 0, goes from character position 0 to 2, has text uint, has token type 'uint', is on line 1(from 1), and is at character position 0. In the second stage, parser would use Solidity rules to verify syntax and build a parse tree, which is used to record how the parser identify the input sentence structure. Figure 4 represents the basic data flow of this case uint token = 1 in which the 'uint' is recognized as a *elementaryTypeName*, the 'token' is an identifier, and the number '1' is classified as *decimalNumber*. These tokens can constitute a sentence, which belongs to a *stateVariableDeclaration*. In this way, we can transform a smart contract source code into a syntax tree.

5.2.2 | Core detector

NeuCheck uses Core detector implemented by dom4j[38] to analyze the vulnerabilities including but not limited to what we discussed in Section 3. As an upgrade product of JDOM, dom4j makes immense improvements in the areas of flexibility, ease of use, and performance. For instance, it can easily process XML, XPath, and XSLT on the Java platform, and fully support DOM, SAX, and JAXP. In this paper, as shown in Algorithm 1, the dom4j parsing process can be stated as follows: (1) create a parser using *SAXreader()*; (2) get *xmlFileStream* from the xml file; (3) get the *targetDocument* object by *read()* method; (4) get the *detectionPattern*; and (5) get the *nodeList* object by *selectNode* method. In the following, we will provide examples of how to build vulnerability detection patterns.

Algorithm 1 Parsing contract XML file by dom4j

input: *vulnerabilityPattern*
output: *nodeList*
1: **function** ParsingFile
2: *saxReader* = SAXReader()
3: *xmlFileStream* = XMLFileReader.read(*"ContractXmlFile.xml"*)
4: *targetDocument* = saxReader.read(*xmlFileStream*)
5: *detectionPattern* = patternReader.read(*vulnerabilityPattern*)
6: *nodeList* = *targetDocument*.selectNode(*detectionPattern*)
7:**end function**

Algorithm 2 Detecting access control vulnerability

```
1: function AccessControl
2:   nodeList ← ParsingFile(AccessControlPattern)
3:   for each node in nodeList do
4:       nodeValue = node.getStringValue()
5:       if nodeValue.equals("public") or nodeValue is NULL then
6:           Document Access Control Vulnerability
7:       end if
8:   end for
9: end function
```

Access control: For such vulnerabilities, we can check whether the functions in the smart contract correctly use public, private, and other keywords for visibility modification. Considering that the constructor usually initializes the smart contract key fields, the contract constructor should be of particular concern. In addition, in order to avoid unauthorized use, it is necessary to check whether the contract is correctly defined and uses the modifier to restrict access to key functions. In this, as shown in Algorithm 2, we define three detection patterns to analyze the vulnerabilities; one pattern analyzes the smart contract constructor, one pattern analyzes the modifiers of functions, and the last pattern analyzes the modifiers of variables.

Hash collision: For such vulnerabilities, we check whether the variable declared in the smart contract contains multiple maps, and the objects stored in these maps are a structure, or verify whether the variable contains both map and variable length arrays. Based on this, we define two detection patterns and, in Section 3, we discussed the principle of generating a hash address by the sha3 method in Solidity. The process is shown in Algorithm 3.

Reentrancy: Through the Solidity parser, we can obtain the intermediate representation and then search if there is a *call()* function. If success, we further get the parameter state after calling the *call()* function. We only care about those *call()* function that does not set gaslimit. Because the reentrancy attack is a function that calls the attacked contract in a loop, it consumes gas for each call. If we find a *call()* function that does not have a gaslimit set, we will treat it as a taint and analyze the current node's remaining instructions and the next nodes whether there is a modify state variable statement. If the smart contract changes its account state after the *call()* is executed, we can determine that the smart contract has a reentrancy vulnerability. The process is shown in Algorithm 4. In view of the basic principle of reentrancy attacks, we recommend that developers use the Checks-Effects-Interactions model. First, the account balance is checked; second, the account balance operation occurs (such as increasing or reducing the balance); and third, the transfer operation proceeds. At the same time, we should use the *transfer()* function in the transfer operation instead of the *call()* function.

Algorithm 3 Detecting hash collision vulnerability

```
1: function Hash Collision
2:   nodeList ← ParsingFile(HashCollisionPattern)
3:   mapNum = 0
4:   vlaNum = 0    //vla means variable-length arrays
5:   for each node in nodeList do
6:       nodeValue = node.getStringValue()
7:       if nodeValue.equals("mapping") then
8:           mapNum++
9:       end if
10:      if nodeValue.equals("vla") then
11:          vlaNum++
12:      end if
13:  end for
14:  if mapNum>1 or ( mapNum==1 and vlaNum>=1 ) then
15:      Document Hash collision Vulnerability
16:  end if
17: end function
```

Algorithm 4 Detecting reentrancy vulnerability

```
1: function Reentrancy
2:   nodeList ← ParsingFile(ReentrancyPattern)
3:   for each node in nodeList do
4:       nodeValue = node.getStringValue()
5:       if nodeValue.equals("extrenalFunctionCall") then
6:           if Assignment exists in its following-sibling-nodes then
7:               Document Reentrancy Vulnerability
8:           end if
9:       end if
10:  end for
11: end function
```

Integer overflow: Here, we give the detection patterns of overflow and underflow. Overflow. Its analysis pattern can be stated as follows: first, obtain the smart contract intermediate representation and search for operation instructions in its node, such as the + instruction; second, obtain the parameters op_0 and op_1 to construct the expression $op_0 + op_1$; and third, detection occurs to determine whether there is an expression *require*$(op_0 + op_1 > op_0)$ in the remaining statement of the current node and following nodes. If the expression does not exist, we will treat it as a vulnerability. If the operation expression is $op_0 + op_1$, we will detect whether there is an expression *require*$(op_0 * op_1 / op_1 == op_0)$ in the remaining statement of the current node and subsequent nodes. Underflow. Its analysis pattern can be stated as follows: first, obtain the smart contract intermediate representation and search for the operation instructions in its node, such as the − instruction; second, obtain the parameters op_0 and op_1 to construct the expression $op_0 - op_1$; third, detect whether there is an expression *require*$(op_0 >= op_1)$ in the remaining statement of the current node and following nodes. If it does not exist, we will treat it as a vulnerability. If the operation expression is op_0 / op_1 or op_0 % op_1, we will detect if there is an expression *require*$(op_1 != 0)$ in the remaining statement of the current node and following nodes.

Solidity is an object-oriented programming language for writing smart contracts. *require()* is a standard library function in Solidity, which is used to verify whether the condition in the parentheses is true. If the condition is not met, an exception will be thrown. When the smart contract is used for calculations, it is necessary to use *require()* to verify the result of the operation. The process is shown in Algorithm 5.

Dependence on predictable variables: For such vulnerabilities, we check for predictable state variables in variable declaration expressions and various assignment expressions. We treat contracts that meet these conditions as vulnerability contracts. The process is shown in Algorithm 6.

5.3 | Case study

We provide an example to demonstrate how NeuCheck detects vulnerabilities in smart contracts. Suppose that the following smart contract contains an access control vulnerability. NeuCheck uses the Solidity parser to transform the smart contract source code into an XML parse tree. The XML parse tree is shown in Figure 5. It is obvious that the root node of the parse tree is *sourceUnit*, which represents the beginning of the contract. The two child nodes of the root are *contractDefine* and *EOF*. The *contractDefine* node represents the declaration of the smart contract, whereas the *EOF* represents the end of the contract. We enter the main part of the contract from the *contractDefine* node, including declaring variables, declaring functions, calling functions, and assigning values to variables.

```
contract Wallet{
    address private owner;
    function Wallet(){
    owner = msg.sender;
     }
    function collectAllFees() public{
    owner.send(collectedFees);
    }
}
```

Algorithm 5 Detecting IntegerOverflow vulnerability

```
1: function IntegerOverflow
2:   statementList ← ParsingFile(IntegerOverflowPattern)
3:   for each statement in statementList do
4:     if (statement is a operation statement) then
5:       op0 = statement.getElement("identifier")
6:       Operator = statement.getElement("expression")
7:       op1 = statement.getElement("identifier")
8:     if require function exists in its following-sibling statement then
9:       callArguments = requiresStatement.getElement("callArguments")
10:      if operator is "+" then
11:        if callArguments does not match "op0 + op1 > op0" then
12:          Document IntegerOverflow Vulnerability
13:        end if
14:      end if
15:      if operator is "*" then
16:        if callArguments does not match "op0 * op1 / op1 == op1" then
17:          Document IntegerOverflow Vulnerability
18:        end if
19:      end if
20:      if operator is "-" then
21:        if callArguments does not match "op1 >= op0" then
22:          Document IntegerUnderflow Vulnerability
23:        end if
24:      end if
25:      if operator is "/" || "%" then
26:        if callArguments does not match "op0 != 0" then
27:          Document IntegerUnderflow Vulnerability
28:        end if
29:      end if
30:    end if
31:  end for
32: end function
```

Algorithm 6 Detecting dependence on predictable variables vulnerability

```
1: function DependenceOnPredictableVariables
2:   nodeList ParsingFile(DependenceOnPredictableVariablesPattern)
3:   for each node in nodeList do
4:     nodeValue = node.getStringValue()
5:     //predictableVars include: timestamp, number, coinbase, gaslimit
6:     if nodeValue.equals("block.predictableVars") then
7:       Document DependenceOnPredictableVariables Vulnerability
8:     end if
9:   end for
10: end function
```

The Solidity parser divides the statement into parts before parsing the smart contract. The semicolon or curly braces can be viewed as the division criteria, and the divided parts are then parsed. Here, we do not illustrate each statement while using a example of state variable declaration statement address private owner; to describe how to construct syntax tree, the owner variable is a state variable that has been declared in the contract. For further parsing, the variable type is

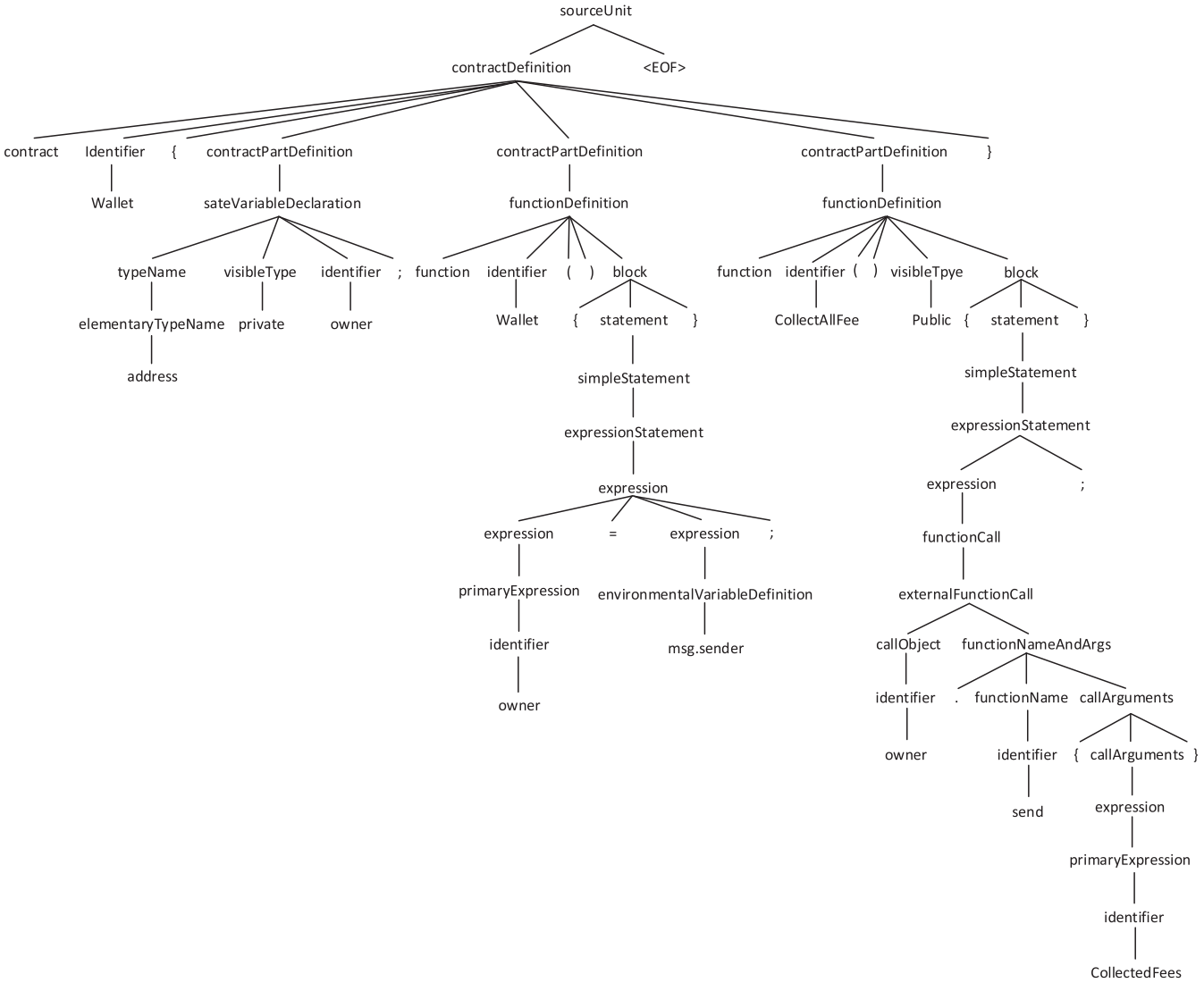

FIGURE 5 Syntax parse tree

the primary type, and the type name is the address. For this variable, its access control type is private, its variable name is owner, and its statement ends with a semicolon. The parse tree clearly represents the structure of the contract and would not cause missing semantics. In fact, NeuCheck transforms the source code of the smart contract into an XML file. Here, to better illustrate the detection process, we replace the XML file with a parse tree. In addition, we then design the detection patterns for access control vulnerability:

```
pattern1 :// functionDefinition[identifier[text()[1] = //contractDefinition/identifier]][visibleType[text()[1]="
    public"] or not(visibleType)]
pattern2 :// contractPartDefinition/functionDefinition[not(visibleType)]
pattern3 :// contractPartDefinition/stateVariableDeclaration[not(visibleType)]
```

where pattern 1 indicates that the control modifier of the smart contract constructor is public or the default, pattern 2 indicates that the control modifier of the smart contract function is the default state, and pattern 3 indicates that the control modifier of the smart contract variable is the default state. The three detection patterns are logical or relational. If the syntax parse tree matches any of the detection patterns, NeuCheck would return an access control vulnerability.

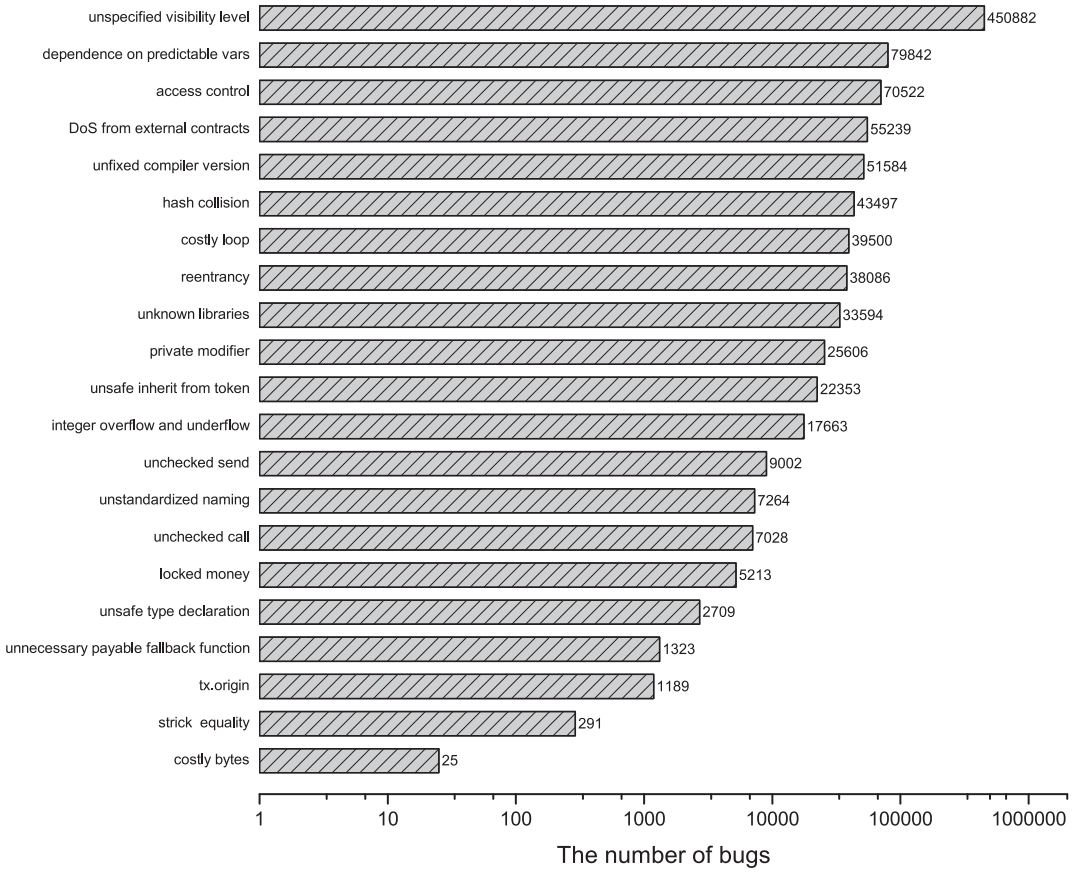

FIGURE 6 Types and proportions of vulnerabilities in smart contracts

6 | EVALUATION

We measure the efficiency of NeuCheck by quantitative analysis and qualitative analysis. We use the web crawler[†] to collect data from more than 52 000 smart contracts from the real Ethereum network since March 24, 2016. We use NeuCheck to analyze these smart contracts for two purposes. We detect the prevalence of security vulnerabilities in real Ethereum smart contracts. In addition, we verify the reliability of NeuCheck because our design and implementation are driven by the characteristics of real-life smart contracts.

6.1 | Quantitative analysis

We use NeuCheck to analyze smart contracts data set both to proof the prevalence of the security vulnerabilities discussed in Section 4 and to help Ethereum developers understand the common vulnerabilities in smart contracts.

Performance. NeuCheck found that more than 99% of smart contracts have varying degrees of security vulnerabilities, and only less than 1% of smart contracts are safe. Figure 6 shows our experimental results, we summarized the types and proportions of Ethereum smart contract vulnerabilities. NeuCheck found a total of 962 412 vulnerabilities. From the figure, we can find that the number of unspecified visibility-level vulnerabilities is the largest among all vulnerabilities, although the vulnerability is not fatal. The number of dependence of predictable variables vulnerability is the second largest, and the third largest is access control vulnerability, which is recorded by us first. The numbers of hash collision and reentrancy vulnerabilities are 43 497 and 38 086, respectively. The number of integer overflow and underflow vulnerabilities is 17 663, ranking 12th out of all vulnerabilities.

[†]The source code has been uploaded to the free source code hosting platform called GitHub, and it is available at https://github.com/Northeastern-University-Blockchain/EthCodeSpider.

The experimental indicates that the security vulnerabilities discussed in Section 4 prevalently exist in the real Ethereum smart contract. These information can help developers discover the potential security risks and standardize the code style improving the security of smart contracts.

6.2 | Qualitative analysis

In this section, we evaluate the performance of NeuCheck by measuring accuracy and time overhead.

6.2.1 | Accuracy

In this section, we will measure accuracy by comparing NeuCheck with two well-konwn smart contract analysis tools, Securify[15] and Mythril.[13] We use three tools to detect the same smart contract data set and compare them to the manual verification results. We define the problems identified by manual verification as true security vulnerabilities, and the problems discovered by the analysis tools are manually labeled as True Positive and False Positive. False Negative is considered a true security vulnerabilities not detected by the tool.

For each analysis tool, the false discover rate (FDR) is the number of false positives divided by the total number of vulnerabilities detected by the tool, ie, FDR=FP/(FP+TP), and the false negative rate (FNR) is the number of false negatives divided by the number of real vulnerabilities in the smart contract (manually verified), ie, FNR=FN/(FN+TP). Smart contract data set includes The Basic Attention Token contract[39] (a new token oriented to the interests of Internet users is integrated with the Brave Web browser); OmiseGO contract[40] (OmiseGO achieves financial compatibility and interoperability through a public, decentralized OMG network); SingularityNET contract[41] (SingularityNET allows anyone to create, share, and profit from artificial intelligence services on a large scale); and Storj contract[42] (Storj provides a low-cost, data-safe privacy protection distributed cloud storage service).

Performance. Table 2 shows the false positive rate and FNR for the three analysis tools. From the data in the table, the average false positive rate of each security analysis tool was NeuCheck 15.725%, Securify 16.675%, and Mythril 60%. The average FNR was NeuCheck 16.525%, Securify 72.75%, and Mythril 83.475%. The experimental results show that NeuCheck keeps the false positive rate and FNR at a low level. Different platforms and domains have different requirements for security analysis tools. According to the characteristics of smart contracts in Section 2.2, the false positive rate and FNR are important indicators for evaluating the detection accuracy of detection tools. Missing any vulnerability can lead to catastrophic consequences, and a relatively high false positive rate can make screening results difficult.

6.2.2 | Time overhead

We measure the time overhead of NeuCheck by comparing it with other analysis tools. We simply divide smart contracts into three categories based on the functionality of smart contracts: financial product contract, game contract, and tool contract. Then, we randomly select 50 contracts for each type of smart contract in the data set. We check the 150 smart contracts by running NeuCheck, Mythril, and Securify and summarize the time overhead and vulnerability categories of each analysis tool.

Performance. The experimental results are shown in Figure 7 and Figure 8. In Figure 7A, we can find that the analysis speed of Securify is slower than Mythril in the previous stage, but Securify's analysis speed exceeds Mythril in the later

TABLE 2 Detection accuracy comparison table

Contract Name		Mythril	Securify	NeuCheck
Basic Attention Token[39]	TP/FP/FN	3/2/5	1/2/7	5/2/3
	FDR(%)	40(%)	66.7(%)	28.6(%)
	FNR(%)	62.5(%)	87.5(%)	37.5(%)
OmiseGO[40]	TP/FP/FN	2/0/5	2/0/5	6/1/0
	FDR(%)	0(%)	0(%)	14.3(%)
	FNR(%)	71.4(%)	71.4(%)	0(%)
SingularityNET[41]	TP/FP/FN	0/1/4	1/0/3	4/1/0
	FDR(%)	100(%)	0(%)	20(%)
	FNR(%)	100(%)	75(%)	0(%)
Storj[42]	TP/FP/FN	0/1/7	3/0/4	5/0/2
	FDR(%)	100(%)	0(%)	0(%)
	FNR(%)	100(%)	57.1(%)	28.6(%)

Abbreviations: FDR, false discover rate; FN, false negative; FNR, false negative rate; FP, false positive; TP, true positive.

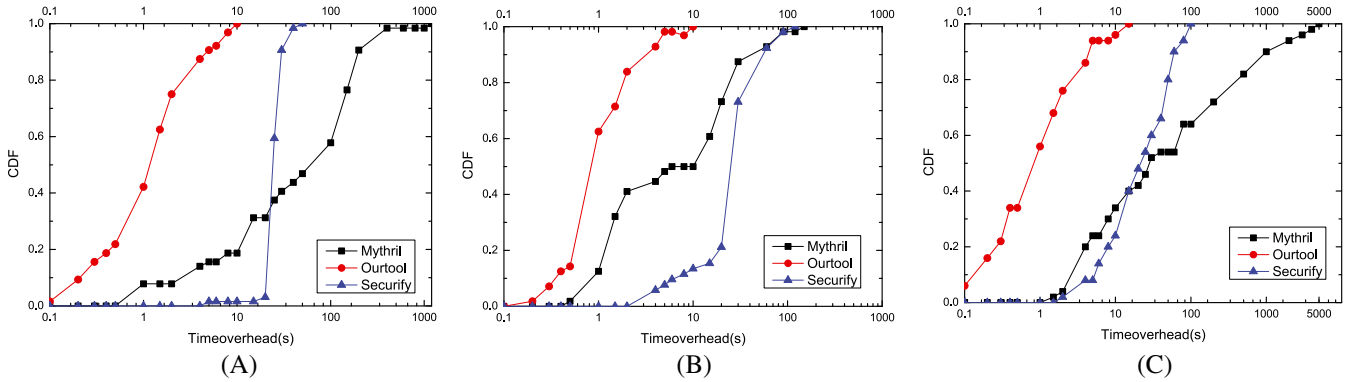

FIGURE 7 The time overhead of each analysis tool. A, Financial product smart contracts analysis result; B, Game smart contracts analysis result; C, Tool smart contracts analysis result [Colour figure can be viewed at wileyonlinelibrary.com]

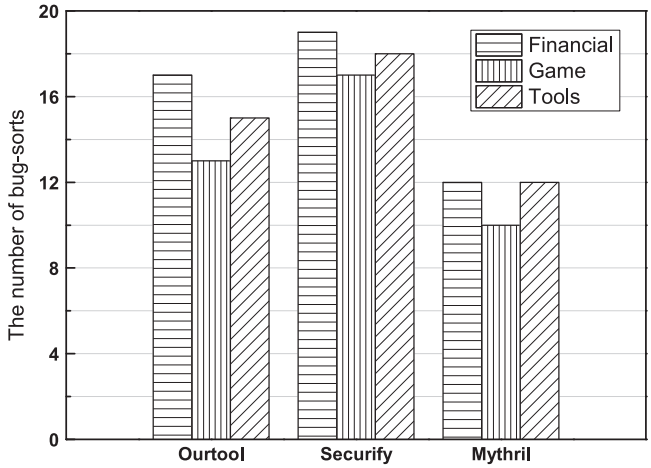

FIGURE 8 The number of vulnerabilities detected by each detection tool

stage. As shown in Figure 7B, Securify and Mythril complete the analysis almost simultaneously. Figure 7C indicates that the analysis speeds of Securify and Mythril are almost equal in the previous stage but that Securify is faster than Mythril in the later stage. However, the time overhead of our tool is minimal regardless of the type of contract. This performance is due to the lightweight system architecture and our optimization of the detection patterns.

From Figure 8, we can find that Securify can detect the largest number of contract vulnerabilities, followed by our tool and lastly by Mythril. NeuCheck can still detect a wide range of vulnerabilities when the analysis speed is greatly improved. Meanwhile, we also find that the number of vulnerabilities detected from financial contract is the largest, followed by the tool contract, and the least is the game contract. This may indicate that we should pay more attention to the security of financial contracts. There may be coincidences in the selection of smart contracts, which affects the number of vulnerabilities detected by each tool, but it does not affect the measure of time overhead.

7 | CONCLUSION

In this paper, we first introduce five new security vulnerabilities in Ethereum smart contracts, which can cause losses worth around millions of cryptocurrencies. For this, we design a more practical Ethereum smart contract security analysis tool termed NeuCheck, which is responsible for verifying the correctness and fairness of contracts. NeuCheck builds the syntax tree so as to transform the source codes to intermediate representation, and then leverage dom4j to parse this tree. Our evaluation with over 52 000 smart contracts indicates that our documented new vulnerabilities are prevalent. NeuCheck is sound and significantly outperforms Securify and Mythril for contracts in our data set in terms of analysis speed and easy deployment.

ACKNOWLEDGEMENT

This work is supported by the National Nature Science Foundation of China (61601107, U1708262, and 61472074), the Fundamental Research Funds for the Central Universities (N172304023), and the China Postdoctoral Science Foundation (2019M653568).

ORCID

Ning Lu https://orcid.org/0000-0001-7325-7307
Bin Wang https://orcid.org/0000-0003-0547-2202

REFERENCES

1. Nakamoto S. Bitcoin: a peer-to-peer electronic cash system. 2008.
2. Ethereum Foundation. Ethereum's white paper. 2014. https://github.com/ethereum/wiki/wiki/White-Paper
3. Dice2win. 2018. http://blogs.360.cn/post/Fairness_Analysis_of_Dice2win_EN.html
4. King of Ether. 2016. https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contractsKingOfTheEtherThrone.sol
5. 'Accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet. 2017. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html
6. TheDAO. 2016. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413
7. Siegel D. Understanding the DAO hack. 2016. https://www.coindesk.com/understanding-dao-hackjournalists/
8. King JC. Symbolic execution and program testing. *Commun ACM*. 1976;19(7):385-394.
9. Hirai Y. Defining the Ethereum virtual machine for interactive theorem provers. Paper presented at: International Conference on Financial Cryptography and Data Security; 2017; Sliema, Malta.
10. Luu L, Chu D-H, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; 2016; Vienna, Austria.
11. Kalra S, Goel S, Dhawan M, Sharma S. Zeus: analyzing safety of smart contracts. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS); 2018; San Diego, CA.
12. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO); 2004; San Jose, CA.
13. Mythril. 2018. https://github.com/ConsenSys/mythril
14. Nikolić I, Kollur A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18); 2018; San Juan, PR.
15. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M. Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18); 2018; Toronto, Canada.
16. Li X, Liu S, Wu F, Kumari S, Rodrigues JJPC. Privacy preserving data aggregation scheme for mobile edge computing assisted IoT applications. *IEEE Internet Things J*. 2018;6:4755-4763.
17. Banerjee M, Lee J, Choo KKR. A blockchain future to Internet of Things security: a position paper, Digital Communications and Networks (2017). http://www.sciencedirect.com/science/article/piiS
18. Xu J, Wang S, Bhargava BK, Yang F. A blockchain-enabled trustless crowd-intelligence ecosystem on mobile edge computing. *IEEE Trans Ind Inform*. 2019;15:3538-3547.
19. Baliga A. Understanding blockchain consensus models. *Persistent*. 2017.
20. Castro M, Liskov B. Practical Byzantine fault tolerance. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI); 1999; New Orleans, LA.
21. Szabo N. The idea of smart contracts. 1997. http://szabo.best.vwh.net/smart_contracts_idea.html
22. Delmolino K, Arnett M, Kosba AE, et al. Step by Step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. Paper presented at: International Conference on Financial Cryptography and Data Security; 2016; Christ Church, Barbados.
23. Vitalik B. Thinking about smart contract security. 2016. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/
24. Moura LD, Nikolaj B. Z3: an efficient SMT solver. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems; 2008; Budapest, Hungary.
25. https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d
26. Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16); 2016; Vienna, Austria.
27. Swamy N, Hriţcu C, Keller C, et al. Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16); 2016; St. Petersburg, FL.
28. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. Smartcheck: static analysis of Ethereum smart contracts. In: Proceedings of the 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB); 2018; Gothenburg, Sweden.
29. An in-depth look at the Parity Multisig Bug. 2017. http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug
30. Rubixi smart contract on Etherscan. https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be

31. Reentrancy. https://consensys.github.io/smart-contract-best-practices/known_attacks/#reentrancy
32. Dworkin MJ. SHA-3 standard: permutation-based hash and extendable-output functions. No. Federal Inf. Process. Stds.(NIST FIPS)-202; 2015.
33. Exception on overflow. 2016. https://github.com/ethereum/solidity/issues/796#issuecomment-253578925
34. SafeMath. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol
35. The run smart contract. https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18
36. Parr T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf; 2013.
37. Aho AV, Sethi R, Ullman JD. Compilers, principles, techniques. *Addison Wesley*. 1986;7(8):9.
38. Yi-feng Z. Research on the analysis of DOM4j tech-nology. *Modern Computer*. 2011;15:39-42.
39. Basic attention token. https://www.stateofthedapps.com/zh/dapps/basic-attention-token
40. OmiseGO. https://www.stateofthedapps.com/zh/dapps/OmiseGO
41. SingularityNET. https://www.stateofthedapps.com/zh/dapps/SingularityNET
42. Storj. https://www.stateofthedapps.com/zh/dapps/Storj

How to cite this article: Lu N, Wang B, Zhang Y, Shi W, Esposito C. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Softw: Pract Exper*. 2019;1–20. https://doi.org/10.1002/spe.2745