Hindawi
Security and Communication Networks
Volume 2021, Article ID 5798033, 12 pages
https://doi.org/10.1155/2021/5798033

Research Article

A Novel Machine Learning-Based Analysis Model for Smart Contract Vulnerability

Yingjie Xu, Gengran Hu, Lin You, and Chengtang Cao

School of Cyberspace, Hangzhou Dianzi University, Hangzhou, China

Correspondence should be addressed to Gengran Hu; gengran.hu@gmail.com and Lin You; mryoulin@gmail.com

Received 29 April 2021; Revised 24 June 2021; Accepted 22 July 2021; Published 15 August 2021

Academic Editor: Abdelouahid Derhab

In recent years, a lot of vulnerabilities of smart contracts have been found. Hackers used these vulnerabilities to attack the corresponding contracts developed in the blockchain system such as Ethereum, and it has caused lots of economic losses. Therefore, it is very important to find out the potential problems of the smart contracts and develop more secure smart contracts. As blockchain security events have raised more important issues, more and more smart contract security analysis methods have been developed. Most of these methods are based on traditional static analysis or dynamic analysis methods. There are only a few methods that use emerging technologies, such as machine learning. Some models that use machine learning to detect smart contract vulnerabilities cost much time in extracting features manually. In this paper, we introduce a novel machine learning-based analysis model by introducing the shared child nodes for smart contract vulnerabilities. We build the Abstract-Syntax-Tree (AST) for smart contracts with some vulnerabilities from two data sets including SmartBugs and SolidiFI-benchmark. Then, we build the Abstract-Syntax-Tree (AST) of the labeled smart contract for data sets named Smartbugs-wilds. Next, we get the shared child nodes from both of the ASTs to obtain the structural similarity, and then, we construct a feature vector composed of the values that measure structural similarity automatically to build our machine learning model. Finally, we get a KNN model that can predict eight types of vulnerabilities including Re-entrancy, Arithmetic, Access Control, Denial of Service, Unchecked Low Level Calls, Bad Randomness, Front Running, and Denial of Service. The accuracy, recall, and precision of our KNN model are all higher than 90%. In addition, compared with some other analysis tools including Oyente and SmartCheck, our model has higher accuracy. In addition, we spent less time for training .

1. Introduction

In recent years, blockchain technology has attracted more and more attention. Blockchain technology represents a fully distributed public ledger and a peer-to-peer platform that makes use of cryptography to securely host applications, transfer digital currencies, messages, and store data [1]. Bitcoin, proposed by Nakamoto in 2008 [2], is the representative blockchain application in the early stage [3]. The stage is often called Blockchain 1.0. Ethereum, which represents the blockchain application of the next stage, is what we call blockchain 2.0. Compared with Bitcoin, Ethereum proposed a novel technology named smart contract. A smart contract is the core of Ethereum, which enables developers to program their own applications in an immune and low-cost manner on the basis of blockchain structure [4]. They are automatically enforced by the consensus mechanism of the blockchain without relying on a trusted authority [4]. Once the smart contract was developed on Ethereum, it could not be changed. Everyone can get the source code of smart contracts developed in Ethereum. Because of its' openness, the smart contract has become the target of many attackers. Since the first Ethereum block was excavated in 2015, a large number of security incidents have occurred. The most famous attack event was the DAO attack [5] in June 2016, which stole nearly 3.5 million ethers from Ethereum. As shown in Table 1, there are so many vulnerabilities of smart contracts for Ethereum. So before smart contracts are released in the blockchain system, it is very important to find

Table 1: DASP TOP 8.

Num	Vulnerability category	Security event
1	Re-entrancy	THE DAO
2	Arithmetic	BeautyChain
3	Access control	ICON
4	Denial of services	GovernMental
5	Bad randomness	SmartBillions lottery
6	Front running	ERC-20
7	Unchecked low level calls	King of the ether
8	Short address	Unkown exchanges

vulnerabilities existing in smart contracts in advance for the security of the blockchain system.

For protecting and checking the security of the smart contract, some code analysis methods were proposed. These methods can mainly be divided into two categories: static analysis methods and dynamic analysis methods. Common dynamic analysis methods include dynamic fuzzing, taint tracking, and so on. Traditional static analysis methods include data flow analysis, static symbolic execution, etc. Fuzzing is a technique for automatically and quickly generating test inputs and running them against a target program to uncover security vulnerabilities [6]. Symbolic execution is a natural extension of normal execution, providing the normal computations as a special case [7]. Dynamic analysis methods need to execute smart contracts in the real blockchain system. The method requires expert knowledge of smart contracts, such as the syntax and semantics of smart contracts. The process of dynamic analysis is very complex because it needs to interact with the real-time blockchain environment. Static analysis methods do not need to run the smart contracts in the real blockchain system. These methods just need source code or bytecode of the smart contract. The static analysis uses a compliance pattern to check the vulnerability of a contract [8]. In order to analyze smart contracts more accurately, static analysis methods often need to provide a lot of matching rules for their pattern. Therefore, both dynamic analysis and static analysis are time-consuming.

In this paper, we propose a model that uses machine learning to analyze the smart contract. Because we get AST (abstract-syntax-tree) from solidity source code, we do not need to execute the smart contract in the real Ethereum system, which is similar to the static analysis method. But we do not make a lot of rules or patterns to check the source code. This method extracts some features from the AST of the smart contract. We will introduce these features and the process of extracting them in Section 3. Then, we use these features to build our machine learning model by applying some machine learning classification algorithms such as KNN [9] and so on. Once we have built our model, we can use it to predict whether the smart contract has the vulnerability. The process of predicting is quicker than the static analysis method and dynamic analysis method. Some tools such as SoliAudit [10] extract features from opcodes directly and then combine machine learning with code analysis to detect smart contract's vulnerabilities. There are still some models that draw features from the AST and CFG (control-flow-graph) and use machine learning for smart contract security analysis [11]. When using these methods, some appropriate feature attributes need to be chosen manually by observing the statistical rules of data they collect. We propose different ways to extract features using shared child nodes. We firstly collect some smart contracts with vulnerabilities. Then, we collect a lot of smart contracts with Etherscan, from which we can get the solidity source code in the real Ethereum. Finally, we compare the structural similarity [12] between ASTs of the smart contract with vulnerability and that of smart contract to test in Etherscan to determine whether there are vulnerabilities in these smart contracts we want to analyze. So we do not need to choose some features manually. Our method predicted many smart contracts' vulnerabilities with overall accuracy, recall, and precision of 90%, which is much higher than that of some common smart contract analysis tools such as Oyente and SmartCheck.

The remainder of this paper is organized as follows. Section 2 introduces the overview of smart contract security. We describe the methodology and implementation of our method in Section 3. Subsequently, in Section 4, we present the result of our experiment and evaluate the accuracy, precision, and recall of the method. Then, we list the common analysis models and compare them with our novel model in Section 5. Finally, we conclude the paper in Section 6.

2. Background

In this section, we introduce the background and overview of smart contract security. Firstly, we list some famous vulnerable Solidity codes and the causes of these vulnerabilities. Then, we offer some existing tools that can analyze the security of smart contracts.

2.1. Vulnerabilities of Smart Contract. Firstly, we will introduce some famous vulnerabilities of the smart contract and the cause of them. Since now Solidity is the most common language used to write smart contract, we mainly present some smart contract written in Solidity. Many of these vulnerabilities are described in Table 1.

2.1.1. Re-Entrancy. Re-entrancy is the primary culprit of the DAO attack. Figure 1 gives an example of the solidity source code with Re-entrancy vulnerability. We call this contract A. The cause of re-entrancy is in Line 11 and Line 12. Firstly, **msg.sender.call.value** is a function by which a smart contract can send ETH to the callers' address (msg.sender) that invokes the withdraw function. But this function is more vulnerable compared to **transfer** (another function that can send ETH from smart contract to the invoker). It does not limit the value of the gas that is generated in the process of the transaction. So we can take advantage of the fallback function to start our attack. Figure 2 shows the corresponding attack smart contract of the example in Figure 1, which we call B. The fallback function lies in the 11th line of the code. Every time a smart contract gets ETH from other smart contracts, it will execute the fallback function.

```
contract A{
    mapping(address=>uint) balances;
    function wallet() view returns(uint){
        return address(this).balance;
    }
    function deposite() payable{
        balances[msg.sender]+=msg.value;
    }
    function withdraw() public{
        if(balances[msg.sender]<address(
            this).balance){
            msg.sender.call.value(
                balances[msg.sender])();
            balances[msg.sender] = 0;
        }
    }
}
```

Figure 1: Re-entrancy.

```
contract B{
  address public bankAddr;
  constructor(address _bank) public{
       bankAddr = _bank;
  }
  funcation Attack() public payable{
       Bank bank = Bank(bankAddr);
       bank.deposite.value(msg.value)();
       bank.withdraw();
  }
  function() public payable{
       if (msg.sender == bankAddr){
                Bank bank = Bank(bankAddr)
                    ;
                bank.withdraw();
  }
  }
}
```

Figure 2: Exploitation of the re-entrancy.

Therefore, the attackers invoke **withdraw** function of Figure 1 in their own fallback function. Then, B can use the fallback function to let A send ETH to B until A has no enough ETH.

2.1.2. Arithmetic. As a common high-risk vulnerability, arithmetic referred to integer overflow and integer underflow, which caused the BEC attack with a huge loss [13]. The attack scenario of arithmetic is shown in Figure 3. Line 9 of this attack scenario requires that the Token of their caller must be greater than the value that he wants to transfer to others. Suppose that balances[msg.sender] = 0 and _value = 1, we can get that balances[msg.sender] < _value. But in fact balances[msg.sender] − _valuevalue = 1. We can bypass the rules of Line 9 in Figure 3 easily.

2.1.3. Access Control. Access control [14] issues are widespread in all programs, not just smart contracts. While insecure visibility settings give attackers straightforward

```
pragma solidity ^0.4.18;
contract Token{
    mapping(address=>uint) balances;
    uint public totalSupply;
    function Token(uint _initialSupply){
        balances[msg.sender] =
            totalSupply = _initialSupply;
    }
    function transfer(address _to,uint
        _value) public returns(bool){
        require(balances[msg.sender]-
            _value >= 0);
        balances[msg.sender]-=_value;
        balances[_to]+=_value;
        return true;
    }
}
```

Figure 3: Attack situation of integer overflow.

ways to access a contract's private values or logic, access control bypasses are sometimes more subtle. These vulnerabilities can occur when the smart contracts use the deprecated tx.origin to validate callers, handle large authorization logic with lengthy requirements, and make reckless use of **delegatecall** in proxy libraries or proxy contracts. In the following example shown in Figure 4, the owner represents the administrator. But the **initContract** function is public, which means that everyone can invoke this function. In other words, everyone has the privileges of the administrator.

2.1.4. Denial of Service. Denial of Service is very dangerous in smart contracts. This kind of attack can break the normal operation of smart contracts and even lead to the collapse of smart contracts. Denial of Service will consume the service capacity of smart contracts. Finally, the attacked smart contracts cannot provide services to other users. In the following example of Figure 5 [15], the attacker can set a too large value for the variable largestWinner to end the loop. In this case, this smart contract cannot accept other users' requests and serve them.

2.1.5. Unchecked Low Level Calls. Some of the deeper features of Solidity are the low-level functions call(), callcode(), delegatecall(), and send(). They will not propagate (or bubble up) and will not lead to a total reversion of the current execution. Instead, they will return a Boolean value set to be false, and the code will still run. If the return values of such low-level calls are not checked, it can lead to fail-opens and other unwanted outcomes. As Figure 6 shows, this smart contract uses a **msg.sender.send** function in Line 5. If users transfer ETH to another smart contract that cannot accept ETH, the function of withdrawal will fail. But **msg.sender.send** function will continue to run and return a false Boolean value. This function will cause an incorrect value of etherLeft and affect the function of withdrawing.

```
contract Send{
constructor() public{
    owner = msg.sender;
}
function initContract() public {
  owner = msg.sender;
}
function trans(address _to,uint money){
    require(owner == msg.sender);
    _to.transfer(money);
}
}
```

Figure 4: Example of the access control.

```
function selectNextWinners(uint256
    _largestWinner) {
  for(uint256 i = 0; i < largestWinner, i
      ++) {
    // heavy code
  }
  largestWinner = _largestWinner;
}
```

Figure 5: Example of the king of the ether.

```
function withdraw(uint256 _amount) public
    {
  require(balances[msg.sender] >= _amount
      );
  balances[msg.sender] -= _amount;
  etherLeft -= _amount;
  msg.sender.send(_amount);
}
```

Figure 6: Example of unchecked low level calls.

2.1.6. Bad Randomness. Real Randomness is hard to generate in Ethereum. Some functions and variables that access apparently hard-to-predict values are generally more public than they seem because the sources of randomness are easy to predict. The function **Keccak256** in Line 5 of the code example in Figure 7 [15] can generate a random value according to the variable seed. Though the variable seed is private, it must have been set via a transaction at a certain time. So it is visible on the blockchain.

2.1.7. Front Running. Users can specify higher fees to have their transactions mined more quickly. Since the Ethereum block is public, everyone can see the contents of others' pending transactions. This situation means that if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with a higher fee to preempt the original solution.

2.1.8. Short Addresses. The principle of Short Addresses is that EVM will automatically add 0 to the encoding of parameters with incomplete length. For example, there is a function in Figure 8 that needs to accept two parameters. One is an address and another is a uint type of number. Suppose

```
uint256 private seed;
function play() public payable {
  require(msg.value >= 1 ether);
  iteration++;
  uint randomNumber = uint(keccak256(seed
      + iteration));
  if (randomNumber % 2 == 0) {
    msg.sender.transfer(this.balance);
  }
}
```

Figure 7: Example of bad randomness.

```
function transfer(address to, uint tokens
    ) public returns (bool success)
```

Figure 8: The example of short addresses.

that the value of parameter "to" is 0xab0 and the value of tokens is 1000, which means that this contract will transfer 1000 ether to 0xab0. If we change this address into 0xab (we miss 0 at the end), EVM will fill 0 at the end of the encoding of these parameters. If the normal encoding is ab0001000, the changed encoding will become ab0010000. The change of this address leads to an increase in the transfer amount, but the contract regards the transfer amount as a normal one.

2.2. Security Analysis Tools of Smart Contract. Secondly, we will introduce some security analysis tools for smart contracts. These tools are different in terms of principle and implementation method. Some code analyzers obtain the Abstract-Syntax-Tree (AST) or bytecode of smart contract source code and extract data flow graph, CFG or XML parse tree from the abstract-syntax-tree. By performing these steps, these code analyzers are able to check all execution paths through CFG for potential bugs [16]. Others rely on the execution of the contract, leveraging symbolic execution, taint tracking, and fuzzing to discover vulnerabilities.

2.2.1. SmartCheck. SmartCheck [17] is a static analysis tool for Ethereum smart contracts implemented in Java. It runs lexical and syntactical analysis on Solidity source code. It uses ANTLR and Solidity grammar to generate an XML parse tree as an intermediate representation (IR). This tool detects vulnerability patterns by using XPath queries on the IR. Thus, it provides full coverage: the analyzed code is fully translated to the IR, and all of its elements can be reached with XPath matching. But SmartCheck has its limitations, as the detection of some bugs requires more dynamic analysis methods such as taint analysis or even manual audit, and SmartCheck detects smart contract's vulnerabilities based on the XPath patterns. In other words, XPath patterns decide the accuracy and effectiveness of SmartCheck. But, there are no perfect ways to get the most precise patterns. So this tool might miss some smart contracts with the vulnerability which has no corresponding XPath patterns. However, SmartCheck is quicker than some tools with dynamic analysis methods to find the problems of smart contracts.

2.2.2. Oyente. Oyente [18] is also a static security analysis tool. Oyente is one of the first tools for the analysis and detection of security issues in Ethereum smart contracts. It is developed by Melonport and its code is open-sourced. Oyente uses symbolic execution on EVM bytecode to identify vulnerabilities. Some of Oyente's detection schemes are not perfect, and the vulnerabilities involved are not comprehensive enough. Compared to some other static analysis tools, Oyente needs to cost more time executing smart contracts in the virtual Ethereum environment.

3. Methodology

In this section, we will introduce the steps of our method. As Figure 9 shows, first of all, we need to build abstract-syntax-tree (A) from the smart contracts that we want to analyze. Then, we collect the AST (B) of some malicious source codes (smart contracts with basic vulnerabilities). To extract the feature vector, we need to get the shared child nodes between A and B and transform these child nodes into vectors. Next, we use some analysis tools, including Slither [19] and Ethainter [20], to label these smart vectors. Finally, we use some machine learning classification algorithms, such as K-Nearest Neighbor Classification (KNN) and Stochastic Gradient Descent (SGD [21]), to train our model.

3.1. Building AST of Smart Contracts. AST is an abstract representation of the source code syntax structure. It expresses the grammatical structure of the programming language in a tree-like form, and each node in the tree represents a structure in the source code. Static code analyzers favor AST as it provides rich details about the characteristics of source codes, such as the number of function definitions. The description of smart contract's AST is shown in Figure 10. The ASTs of smart contracts can be parsed in terms of type, name, child nodes, and value. In the meantime, child nodes also can be parsed in these four characters. By the special syntax structure, child nodes are easy to be traversed, which is helpful for us to get the share child nodes between two ASTs. We build the AST from a smart contract by py-solc-x (a python third-party package). With this package, we can transform the source code of smart contracts into AST in the form of json. To handle ASTs better, we change the form of AST to dict (a python data structure that has a key and value).

3.2. Collecting AST of Basic Malicious Smart Contract. First, we introduce the concept of basic malicious smart contract. For example, as stated in Section 2, arithmetic is known as integer overflow and integer underflow. But in fact, it contains many situations.

(i) Basic Principles of Arithmetic

We declare that the data structure of x is **uint8**. We can infer that $0 \le x \le 255$. If $x \leftarrow 255 + 1$, x will become 0. If $x \leftarrow 0 - 1$, x will become 255.

(ii) Multiplication Arithmetic

We define that **uint256**x = amount*_value, while value is an immutable value. We can change the value of the amount to make x bigger than $2^{256} - 1$ or smaller than 0. In this case, we will cause integer overflow and integer underflow.

(iii) Addition Arithmetic

We define that **uint256**x = amount + _value. We also are able to make x exceed its value range in order to lead to addition arithmetic.

(iv) Subtraction Arithmetic

One of the forms of subtraction arithmetic feature codes is x = _value − amount. Modifying the value of amount can beget subtraction arithmetic.

Like Re-entrancy, it also has some attack situations. If **msg.sender.call.value** arises, there will be a high probability of causing re-entrancy. We name smart contracts with these most basic vulnerabilities basic malicious smart contracts. We collect basic malicious smart contracts and classify these malicious smart contracts according to the corresponding vulnerabilities.

3.3. Getting Shared Child Nodes. After we get the AST of the smart contract to be analyzed and the AST of the malicious smart contract, we need to obtain the shared child nodes between both of the ASTs. Specific implementation algorithms are described in Algorithms 1 and 2. Algorithm 1 gets all of the nodes from an AST and Algorithm 2 gets the shared child nodes between both of the ASTs. Algorithm 1 makes AST and an empty list (a python data structure like Array) as input. We call this list_A. We use recursion to traverse all nodes. Because the data type of the AST has three kinds: dict, list, and string, we need to traverse the node according to the data type corresponding to the node. If the data type of the node is list, we traverse all elements of the current node. If the data type of node is dict, we traverse all of the next level nodes of the current node. If the data type of the node is string, we will add the node to list_A. Algorithm 2 takes AST_A and AST_B as input and outputs a list called child_node. AST_A is the AST of the smart contract to be analyzed. AST_B is the AST of the malicious smart contract. First, we get all of the nodes in AST_A and AST_B by Algorithm 1. We make node_A as the collection of all of the nodes in AST_A and make node_B as the collection of all of the nodes in AST_B. Then, we traverse all nodes in node_A. If there is a node in both node_A and node_B, we add this node to the child_node. Finally, we get the shared child node between two ASTs in the form of a list.

3.4. Extracting Feature Vectors. We extract feature vectors from the shared child nodes. First of all, we already have some malicious smart contracts that are classified according to vulnerability categories. Then we obtain the shared child node between the smart contract that we need to judge and each type of malicious smart contracts. Finally, we will get a vector consisting of the number of common child nodes. For example, we have 4 malicious smart contracts with arithmetic. We build a vector $\mathbf{v}_1$ consisting of 4 ASTs.

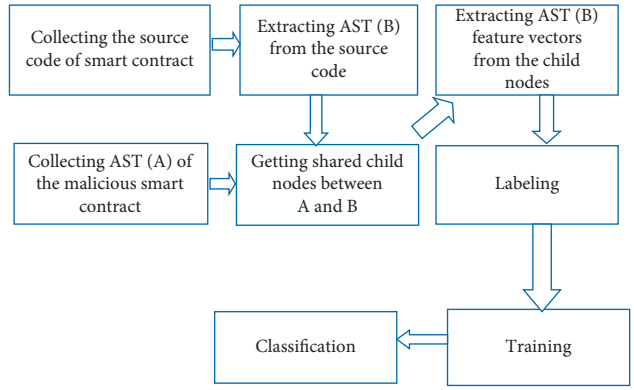

Figure 9: The overview of security analysis based on machine learning.

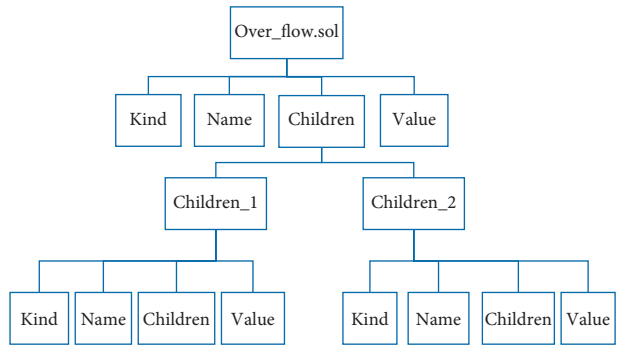

Figure 10: The graph of AST.

Input: ast_A: AST of the smart contract; node_list: an empty list; current_key: the keys of AST;
Output: node_list: all node of AST in the form of list current_key = None

```
if ast_A is list type then
    for key ∈ ast_A.keys do
        Algorithm 1 (i, node_list, current_key)
    end for
else
    if ast_A is dict type then
        for key, value ∈ ast_A.items do
            Algorithm 1 (value, node_list, key)
        end for
    else
        node_list.append ( [current_key, ast])
    end if
end if
```

Algorithm 1: Get all node and node's value from AST.

Input: ast_A: AST of the smart contract to be analyzed ast_B:AST of malicious smart contract
Output: child_node: the list of the shared child node Algorithm 1 (ast_A, $node_A$)Algorithm 1 (ast_B, $node_B$)child_node is an empty list

```
for node ∈ node_A do
    if node ∈ node_B then
        child_node.append (node)
    end if
end for
```

Algorithm 2: Get common child nodes.

$$\mathbf{v}_1 = (\mathrm{AST}_1, \mathrm{AST}_2, \mathrm{AST}_3, \mathrm{AST}_4). \tag{1}$$

Then we will obtain a vector composed of the number of common subnodes between the analyzed smart contract and this vector. Suppose that the number of common subnodes is a, b, c, d, we get a feature vectors $\mathbf{v}_2$.

$$\mathbf{v}_2 = (a, b, c, d). \tag{2}$$

3.5. Labeling. For training our model with feature vectors, we will label our feature vectors. We use some analysis tools, including Slither [19] and Ethainter [20], to judge which vulnerability the smart contract in the data set possesses. Then we label the corresponding vulnerability for these feature vectors. If some smart contracts have no vulnerability, we will label them no vulnerability.

3.6. Training and Classification. We use some machine learning classifying algorithms, such as KNN and SGD, to train our model.

(i) KNN

KNN is one of the simplest methods in data mining classification technology. Firstly, we set a value named **k**. We already have a matrix **M** with m rows and n columns

$$\mathbf{M} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}. \tag{3}$$

Then, we extract a feature vector **b** from our shared subnodes between smart contracts to detect malicious smart contracts.

$$\mathbf{b} = (b_1, b_2, \ldots, b_n). \tag{4}$$

We will calculate the distance between these vectors **b** and each row of the matrix.

$$d_i = \sqrt{\sum_{j=0}^{n} \left(a_{ij} - b_j\right)^2}. \tag{5}$$

Then we will obtain a distance vector **d**

$$\mathbf{d} = (d_1, d_2, \ldots, d_m). \tag{6}$$

Next, we sort **d** and get the smallest first k elements in this vector. The label that appears the most in these k elements will be the label of the analyzed smart contract.

(ii) SGD

Gradient Descent is the process of minimizing a function by following the gradient of the loss function. Suppose that we have a matrix **X** with m rows and n columns and a vector **y** with m elements.

$$\mathbf{X} = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}, \quad \mathbf{y} = (y_1, y_2, \ldots, y_m). \tag{7}$$

All elements in **y** can only be equal to 1 or -1. We want to train a function $\mathbf{f}(\mathbf{X}) = \mathbf{X}\mathbf{w}^T + \mathbf{b}^T$ where **b** is a row vector with m elements and **w** is a row vector with n elements.

$$\mathbf{w} = (w_1, w_2, \ldots, w_n), \quad \mathbf{b} = (b_1, b_2, \ldots, b_m). \tag{8}$$

Loss function is used to modify the value of the **w** and **b**, which is defined as follows:

$$L = \sqrt{\sum_{i=1}^{m} \left(\sum_{j=1}^{n} \left(w_j \cdot x_{ij}\right) + b_i - y_i \right)^2}. \tag{9}$$

We update **w** and **b** by making the L become smaller and smaller. Finally, we find the most suitable function $\mathbf{f}(\mathbf{X}) = \mathbf{X}\mathbf{w}^T + \mathbf{b}^T$ for our training model.

4. Experiment

4.1. Collecting Data Set. The first step of our experiment is to collect enough data. We use three data sets including Smartbugs [22], SolidiFi-benchmark, and Smartbugs-wilds.

(i) Smartbugs

Smartbugs is a dataset consisting of labeled smart contracts according to their vulnerabilities. The concept information about Smartbugs is shown in Table 2. Smart contracts in Smartbugs have the most basic vulnerability. So they are made to create basic malicious smart contract data sets.

(ii) SolidiFi—benchmark

SolidiFi—benchmark also has labeled smart contracts with different kinds of vulnerabilities, which is similar to Smartbugs. But SolidiFi—benchmark has more smart contracts than SmartBugs. The smart contracts in this dataset have been deployed in the real Ethereum network.

(3) Smartbugs—wilds

Smartbugs—wilds have a lot of smart contracts of real Ethereum environment. There are 13241 smart contracts in this dataset in total. But these smart contracts are not labeled. Therefore, we can not directly know whether the smart contract in this dataset is vulnerable and we would use some analysis tools such as Slither and Ethainter to label these smart contracts. If these tools divided the smart contract into the same kinds of vulnerability or normal, the smart contract would be labeled. Otherwise, we abandon this smart contract.

First, we collect basic malicious smart contracts from Smartbugs. After collecting basic malicious smart contracts, we collect a lot of smart contracts as the final experimental dataset to build and evaluate our method for smart contract vulnerability from Smartbugs-wilds and SolidiFI—benchmark. Table 3 shows us the information of the final experimental dataset.

4.2. Implemention. We obtain the ASTs from smart contracts in data sets by two python third-package that are pysolc-x and solidity-parse. First, we obtain the ASTs from basic malicious smart contract dataset and the final experimental smart contract dataset. Then we transfer these ASTs into another form like dict (a python data structure) in order to traversal all of the nodes of the AST easier. Next, we get the shared child nodes between basic malicious smart contracts' ASTs and final experiment smart contracts' ASTs. Afterward, we make use of these shared child nodes to extract some feature vectors. In the meantime, we label these vectors on the basis of corresponding labeled smart contracts in the final experimental dataset and get our last data sets. Finally, we use our machine learning algorithm to build our model and evaluate the effectiveness of our model.

4.3. Evaluation. First, we split the data set consisting of the feature vectors. The train set and test set are split at a ratio of 70% and 30%. There are 286 train samples and 122 test samples. We evaluated the effectiveness of different features and models by accuracy, precision, and recall. A confusion matrix is shown in Table 4. We use this confusion matrix to describe the concept of accuracy, precision, and recall.

(1) Accuracy

Accuracy reflects the ability of the classifier to judge the whole data set:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}. \tag{10}$$

(2) Precision

Precision refers to the accuracy between the predicted value and the real value:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{11}$$

(3) Recall

Recall indicates the ratio of the positive cases of the predicted pair to the true-positive cases:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{12}$$

We build two models for our experiment using KNN and SGD. In order to better show our experimental results, we set up some comparative experiments to compare the effectiveness of our models with some famous security tools such as SmartCheck and Oyente by accuracy, recall, and precision.

Table 2: Date set of malicious smart contracts.

Vulnerability category	The number of smart contracts
Arithmetic	15
Access control	19
Re-entrancy	20
Bad randomness	8
Denial of service	6
Unchecked low level	5
Short address	1
Front running	4

Table 3: Data set of the smart contracts for our experiment.

Vulnerability	The number of smarts in data set
Arithmetic	50
Re-entrancy	60
Access control	48
Bad randomness	50
Denial of service	50
Unchecked low level	50
Short address	50
Front running	50

Table 4: Confused matrix.

		Forecast	
		Yes	No
Actual	Yes	True positive (TP)	False negative (FN)
	No	False positive (FP)	True negative (TN)

4.4. Result. The experimental result is shown in Table 5. As we can see, the KNN model had the best experimental result in detecting vulnerabilities, which had more than 90% accuracy, recall, and precision for all smart contract vulnerabilities in the experiment. The experimental result of the SGD model was not very stable, which had the high accuracy for Re-entrancy, Arithmetic and Access Control, but had the general accuracy for other vulnerabilities. Compared to Oyente and SmartCheck, machine learning models achieved better experimental results, although we think there are some reasons for the situation that Oyente and SmartCheck can not have a good experimental result.

4.4.1. The Comparison of Our Model to Oyente. Oyente is the earliest tool for detecting smart contract vulnerabilities. So Oyente can not detect some vulnerabilities raised in the last two or three years including some vulnerabilities belonging to Access Control, Bad Randomness, Denial of Service, Short Address and Front Running. In other words, Oyente can only detect 7 types of vulnerabilities. Therefore this tool has low experiment scores in the above five kinds of vulnerabilities. In addition, Oyente uses a dynamic analysis method to detect vulnerabilities. So this tool needs an Ethereum environment and solc compiler to execute detecting smart

Table 5: The result of experiment.

Vulnerabilities	Models						Tools					
	KNN			SGD			Oyente			SmartCheck		
Names	Accuracy (%)	Recall (%)	Precision (%)	Accuracy (%)	Recall (%)	Precision (%)	Accuracy (%)	Recall (%)	Precision (%)	Accuracy (%)	Recall (%)	Precision (%)
Re-entrancy	95.45	95.45	95.83	95.45	68.18	70.83	70.10	72.10	72.10	75.20	60.10	63.50
Arithmetic	95.54	90.90	91.10	90.10	98.10	95.10	74.50	62.10	64.50	85.20	53.10	65.54
Access control	95.00	95.00	96.67	95.45	94.44	95.54	50.10	54.10	52.50	67.20	53.10	65.54
Bad randomness	90.10	93.10	91.10	74.10	75.10	73.10	46.10	47.20	47.30	54.10	55.10	56.10
Denial of service	90.20	90.26	90.56	64.20	65.10	62.40	47.10	48.42	43.64	61.10	59.10	60.20
Unchecked low level	90.10	90.10	90.10	81.10	81.20	80.10	65.65	63.34	54.24	59.10	50.20	50.10
Short address	91.10	91.10	92.85	78.10	77.27	82.14	49.30	48.45	47.65	67.10	61.10	61.10
Front running	95.45	95.45	96.10	89.90	86.10	86.20	54.27	56.18	59.26	55.20	56.20	54.10

contracts, which leads to the fact that Oyente is highly dependent on solc compiler versions and Ethereum environment. During our experiment, some smart contracts could not be analyzed because their syntax is incompatible with the solc compiler and Ethereum environment. If smart contracts do not match the version of solc compiler and are not executed in the Ethereum environment, Oyente will not execute these smart contracts normally.

4.4.2. The Comparison of Our Model to SmartCheck. SmartCheck is a static analysis tool for smart contract vulnerabilities. As a result, it does not need to execute smart contracts in the real Ethereum environment. In addition, SmartCheck can analyze more vulnerabilities than Oyente because this tool sets a lot of patterns to match corresponding vulnerabilities. But some patterns are not precise enough, which leads to the low accuracy of this tool to analyze smart contracts.

4.4.3. Analysis of Our Model. From the result of our experiment, we can know that our model based on machine learning classification algorithms has a good result compared to other analysis tools. But we found some problems in our experiment. First, with the exception of Re-entrancy, Arithmetic, Access Control, and Front Running, the experiment result of other vulnerabilities has declined. KNN model and SGD model both have this situation. We find that the number of these basic malicious smart contracts corresponding to these vulnerabilities is so small. The fact may have a certain impact on our experiment. Compared to the KNN model, the SGD model looks so unstable and performs not well. Because the SGD model needs so many parameters, and the model may not perform well if we set some unsuitable parameters for our model. Different from other tools, we just detect smart contract vulnerabilities based on their own characteristics of structure and grammar. So we do not need to execute smart contracts or complex patterns to analyze smart contracts. Therefore, our method based on machine learning and shared child nodes have a better experimental result.

5. Related Work

5.1. Static Code Analysis. Static code analysis uses smart contracts' semantic attributes such as EVM bytecode and assembly code to detect their bugs. Mythril [21] is a static analyzer that detects vulnerabilities through symbolic execution with EVM bytecode. SmartCheck [18] uses pattern matching to analyze smart contracts in the form of XPath. These tools just verify some vulnerabilities that are predefined. In addition, this analysis method needs to fix up the matching rules manually based on their analysis efficiency. So static code analysis has a high false-positive rate.

5.2. Dynamic Analysis Method. The most famous technology that makes use of the dynamic analysis method is formal verification. This method translates the source code or bytecode of the smart contract into some functions and checks the security of all functions by running the smart contract in a real blockchain system. Saxena et al. [23] adopted formal semantics of EVM to verify smart contracts. Compared to static code analysis, the dynamic analysis method executes smart contracts in the real blockchain system, which makes the dynamic analysis method more accurate but more time-consuming.

5.3. Traditional Machine Learning Model. The traditional machine learning model also deals with the semantic characteristic of smart contracts such as bytecode and AST, which is similar to static code analysis. But this model must set feature attributes manually and choose the most suitable feature attributes to build a training model. Compared to static code analysis, this method has a higher true-positive ratio. Compared to the dynamic analysis method, this solution is quicker to analyze smart contracts. But the traditional machine learning based analysis method is required to adjust their feature attributes manually through accuracy, recall, and precision; the process of finding out the most suitable one costs much time. Furthermore, this model is difficult to extend for analyzing some new vulnerabilities because the model will change with different feature attributes, which means that the model will reselect features and retrain the analysis model.

5.4. Our Machine Learning Model. Our model keeps some advantages over the traditional machine learning model. First of all, we achieve more accuracy, recall, and precision than some static analysis tools. In addition, we can analyze smart contracts with faster speed than some dynamic analysis tools. Also, our model updates some advantages based on the traditional machine learning model. By the shared child nodes of ASTs, our model can train the analysis model automatically rather than adjusting the feature attributes manually, and it is also easier to extend the model for detecting new vulnerabilities. As long as we have the corresponding vulnerability smart contract data sets, we can expand our model to find the vulnerability. Detailed comparisons between our model and other solutions are shown in Table 6.

Table 6: The comparison.

Analysis method	Comparison between these analysis methods		
Names	The speed of code analysis	The accuracy of code analysis	Difficulty of training analysis model
Static code analysis	Quick	Low	Difficult
Dynamic analysis method	Slow	High	Difficult
Traditional machine learning-based analysis method	Quick	High	Difficult
Our novel machine learning-based analysis method	Quick	High	Easy

6. Conclusion

In this work, we introduce a model that uses machine learning to analyze smart contracts' vulnerabilities. First, we extract feature vectors from ASTs. Next, we get shared child nodes from the two ASTs. Finally, we use feature vectors to train our machine learning model and build a model which can detect the existed kinds of the smart contracts' vulnerabilities in Ethereum written in Solidity with 90% more for the accuracy, recall, and precision. In terms of the accuracy of detecting smart contracts, our model is better than some tools such as Oyente and SmartCheck. In addition, our model just needs ASTs of the analyzed smart contracts to find vulnerabilities without setting complicated patterns or executing them in Ethereum, and so it is simple for professionals to make use of this model.

However, our model needs to be improved in the following aspects. First, we just detect which kinds of vulnerability the smart contract possesses, but we do not detect concrete problems and we also cannot locate the line of code where the vulnerability occurs in the smart contract. Second, this model just detects smart contracts in Ethereum written in Solidity. In the next step, we want to detect other blockchain systems such as EOS and Fabric. Third, the number of our basic malicious smart contracts is not enough, which will have an effect on the accuracy of the method. Last but not least, we do not make a tool using our model to detect smart contracts.

Data Availability

The data used to support the findings of the study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially supported by the Key Program of the National Natural Science Foundation of China (no. 61772166) and the Natural Science Foundation of Zhejiang Province of China (no. LZ17F020002).

References

[1] D. Chris, *Introducing Ethereum and Solidity Foundations of Cryptocurrency and Blockchain Programming for Beginners*, Apress, New York, NY, USA, 2017.

[2] S. Nakamoto and A. Bitcoin, "A peer-to-peer electronic cash system," vol. 4, 2008 Bitcoin.–URL: https://bitcoin.org/bitcoin.pdf.

[3] W. Ethereum, "A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[4] X. Wang, X. Zha, G. Yu et al., "Attack and defence of ethereum remote apis," in *Proceedings of the 2018 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, IEEE, Abu Dhabi, UAE, December 2018.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Proceedings of the International conference on principles of security and trust*, pp. 164–186, Springer, Thessaloniki, Greece, April 2017, Lecture Notes in Computer Science.

[6] Bo Jiang, Ye Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 259–269, IEEE, Corum, Montpellier, France, September 2018.

[7] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 531–548, Association for Computing Machinery, London, UK, November 2019.

[8] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, pp. 110–115, Association for Computing Machinery, New York, NY, USA, January 2020.

[9] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: a lazy learning approach to multi-label learning," *Pattern Recognition*, vol. 40, no. 7, pp. 2038–2048, 2007.

[10] J. Liao, T. Tsai, C. He, and C. Tien, "Soliaudit: smart contract vulnerability assessment based on machine learning and fuzz testing," in *Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pp. 458–465, IEEE, Granada, Spain, October 2019.

[11] P. Momeni, Y. Wang, and R. Samavi, "Machine learning model for smart contracts security analysis," in *Proceedings of the 2019 17th International Conference on Privacy, Security*

and Trust (PST), pp. 1–6, IEEE, Fredericton, NB, Canada, August 2019.

[12] W. Wen, X. Xue, Y. Li, P. Gu, and J. Xu, "Code similarity detection using ast and textual information," *International Journal of Performability Engineering*, vol. 15, no. 10, Article ID 2683, 2019.

[13] Á. Hajdu and D. Jovanović, "Solc-verify: a modular verifier for solidity smart contracts," in *Proceedings of the Working Conference on Verified Software: Theories, Tools, and Experiments*, pp. 161–179, Springer, New York City, NY, USA, July 2019.

[14] C. F. Torres, M. Baden, R. Norvill, and H. Jonker, "Ægis: smart shielding of smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2589–2591, Association for Computing Machinery, New York City, NY, USA, November 2019.

[15] Ncc Group, "Decentralized application security project top 10," 2018, https://www.dasp.co/.

[16] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*, pp. 375–380, IEEE, Yogyakarta, Indonesia, November 2018.

[17] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pp. 9–16, Association for Computing Machinery, Gothenburg, Sweden, May 2018.

[18] E. Zhou, S. Hua, B. Pi et al., "Security assurance for smart contract," in *Proceedings of the 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, Paris, France, February 2018.

[19] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, IEEE, Montreal, Canada, May 2019.

[20] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 454–469, Association for Computing Machinery, New York City, NY, USA, June 2020.

[21] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," vol. 1, pp. 685–693, in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, vol. 1, pp. 685–693, MIT Press, Cambridge, MA, USA, December 2015.

[22] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 530–541, New York City, NY, USA, July 2020.

[23] M. Saxena, P. Daian, D. Park, Y. Zhang, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium*, pp. 912–915, New York City, NY, USA, November 2018.