Hindawi
Wireless Communications and Mobile Computing
Volume 2023, Article ID 9212269, 12 pages
https://doi.org/10.1155/2023/9212269

Research Article

A Smart Contract Vulnerability Detection Model Based on Syntactic and Semantic Fusion Learning

Daojun Han,[1] **Qiuyue Li**,[2] **Lei Zhang**,[1] **and Tao Xu**[1]

[1]*Henan Engineering Research Center of Intelligent Technology and Application, Henan University, Kaifeng 475004, China*
[2]*School of Computer and Information Engineering, Henan University, Kaifeng 475000, China*

Correspondence should be addressed to Lei Zhang; zhanglei@henu.edu.cn

Received 7 July 2022; Revised 4 November 2022; Accepted 24 November 2022; Published 3 February 2023

Academic Editor: Changyan Yi

Copyright © 2023 Daojun Han et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

As a trusted decentralized application, smart contracts manage a large number of digital assets on the blockchain. Vulnerability detection of smart contracts is an important part of ensuring the security of digital assets. At present, many researchers extract features of smart contract source code for vulnerability detection based on deep learning methods. However, the current research mainly focuses on the single representation form of the source code, which cannot fully obtain the rich semantic and structural information contained in the source code, so it is not conducive to the detection of various and complex smart contract vulnerabilities. Aiming at this problem, this paper proposes a vulnerability detection model based on the fusion of syntax and semantic features. The syntactic and semantic representation of the source code is obtained from the abstract syntax tree and control flow graph of the smart contract through TextCNN and Graph Neural Network. The syntactic and semantic features are fused, and the fused features are used to detect vulnerabilities. Experiments show that the detection accuracy and recall rate of this model have been improved on the detection tasks of five types of vulnerabilities, with an average precision of 96% and a recall rate of 90%, which can effectively identify smart contract vulnerabilities.

1. Introduction

Smart contracts were proposed by Nick Szabo in 1995 and are defined as " a set of promises, specified in digital form, including protocols within which the parties perform on these promises." [1]. A smart contract defines a contract in digital form, and the contract is automatically executed when the contract participants meet the conditions required by the smart contract. Due to the lack of a trusted execution environment, smart contracts were not really implemented before blockchain technology was proposed. In 2008, Satoshi Nakamoto proposed the concept of blockchain in "Bitcoin: A Peer-to-Peer Electronic Cash System", which provides a trusted and immutable execution environment for smart contracts. In 2014, Vitalik Buterin proposed Ethereum, inspired by Bitcoin. Ethereum is the first blockchain platform to support developers in creating smart contracts and decentralized applications. Through smart contracts, developers can create decentralized applications on platforms such as Ethereum to achieve trusted transactions. Compared with traditional contracts, smart contracts can improve transaction efficiency and have the characteristics of decentralization and immutability. Smart contracts provide security methods superior to traditional contracts, but like other programming languages, smart contracts also have some security vulnerabilities. In the 2016, the DAO incident, attackers stole more than 3.6 million ethers by exploiting a vulnerability in the smart contract code. In 2018, there was a major vulnerability in the BEC smart contract. The attacker used the integer overflow problem of the transfer function to generate tokens indefinitely, causing a loss of about 6 billion [2]. Smart contracts involve the digital properties of a large number of users, and loopholes in smart contracts will cause the loss of digital assets. At the same time, the immutable property of the blockchain makes it difficult to repair the contract vulnerability once it occurs, so it is necessary to check the contract vulnerability before deploying the smart contract on the chain.

According to whether the code is executed during detection, the early vulnerability detection methods are mainly

static analysis and dynamic analysis methods [3–6]. Static analysis methods can analyze the complete control flow and data flow of the code and have a high level of code coverage [7, 8]. The dynamic analysis tool executes the target program on a real system or in an emulator that can accurately detect program errors [9, 10]. However, using static analysis and dynamic analysis methods to detect smart contracts need to rely on artificially defined expert rules, which results in low detection efficiency and a long time. In recent years, researchers have begun to use deep learning methods for vulnerability detection. Using deep learning, effective features can be automatically extracted from source code without relying on expert-defined vulnerability features. But, there is a diversity of features in the source code, and when applying deep learning methods for vulnerability detection, it is necessary to consider how to better extract features from the source code of smart contracts. Most current research has focused on single feature representations, with some models using program source code or binary files directly as input to extract key feature information such as identifiers, function types, and operators associated with character streams [11–14]; or converting source code into an abstract syntax tree from which the syntactic features of the source code are extracted [15–18]. The source code of smart contracts contains rich feature information, and a single code representation cannot retain the rich syntactic and semantic information in the source code. When using deep learning methods, it is necessary to consider building an appropriate data representation method to extract the feature information related to vulnerabilities to the greatest extent.

This paper proposes a multifeature fusion vulnerability detection model based on deep learning in order to solve the problem of insufficient code representation ability in the current smart contract vulnerability detection model. The model can better extract the source code features, enhance the code representation ability, and improve the accuracy of smart contract vulnerability detection. The model obtains the syntactic and semantic features of smart contracts through Abstract Syntax Tree (AST) and Control Flow Graph (CFG), and constructs a feature space through syntactic and semantic fusion vectors. The establishment of the model is mainly divided into three steps: First, the Ethereum smart contracts are collected, followed by balancing the training dataset using a weighted random sampling method. Then, the AST and CFG of the smart contract are obtained from the Solidity source code, the static features of the contract are extracted from the AST and CFG. Finally, the syntactic and semantic features are concatenated to obtain the feature fusion vector, as well as to use the fusion vector for vulnerability detection of smart contracts. Fusion features can better characterize the vulnerability-related features in the source code, from which the model learns vulnerability patterns for effective and fast detection of vulnerabilities.

The rest of this article is organized as follows. Section 2 discusses related work. Section 3 introduces the types of smart contract vulnerabilities. Section 4 presents the general framework and detailed steps of the model. Section 5 shows the results of our experiments, evaluating the accuracy, precision, recall, and F1 values of the model. Finally, we conclude this paper in Section 6.

```
1  contract OverflowLoop{
2    uint256 public count;
3
4    function OFloop(uint256[] _array) public {
5         count = 0;
6         for(uint8 i = 0; i < _array.length; i++){
7              count++;
8         }
9    }
10 }
```

Figure 1: Integer overflow vulnerability example.

2. Background

In this section, several types of smart contract vulnerabilities studied in this paper are presented. The vulnerabilities studied in this paper can be divided into two layers: the Solidity code layer and the blockchain system layer. There are many types of vulnerabilities in the Solidity code layer, including reentrancy vulnerability, insecure arithmetic, permission control vulnerability, denial of service vulnerability, and unknown function call vulnerability. The blockchain system layer includes Timestamp Dependency vulnerability, Block Parameter Dependency vulnerability, and Transaction-Ordering Dependence vulnerability [19–21], among others. This paper mainly studies insecure arithmetic, reentrancy, timestamp dependency, and implicit visibility vulnerabilities that are harmful to the blockchain.

2.1. Insecure Arithmetic. Integer overflow and integer underflow vulnerabilities are caused by values that are outside or below the defined range of the integer type. In computer languages, integer type numbers have maximum and minimum values. In blockchain, integers are unsigned numbers in the range of 0 to 255 [22]. If the maximum value of 255 is exceeded, it will overflow and cause a zero return situation. If the value is 0, subtracting 1 will cause the underflow to become the maximum value. The addition and multiplication operations of numbers can cause overflow problems, and the subtraction of numbers can create underflow problems. Figure 1 contains a function that has the risk of integer overflow. If an attacker passes a value of length greater than 255 to the OFloop () function, an integer overflow will occur for uint8 *i*. The loop condition *i* < _array.length is always satisfied, and the loop will keep executing until all the gas fee is consumed.

2.2. Reentrancy. The reentrancy vulnerability is a serious vulnerability. Ethereum smart contracts are able to call and use code from other external contracts to send ether to various external user addresses. The operation of calling an external contract or sending ether to an address requires the contract to submit external calls. An attacker can hijack these external calls to force the contract to execute further code, including calls to itself. If the transfer function uses the call. Value () function to transfer money, the call. Value () function will automatically trigger the fallback () function. If the transfer function modifies the status variable of the

```
1 contract EtherStore {
2
3     uint256 public withdrawalLimit = 1 ether;
4     mapping(address => uint256) public lastWithdrawTime;
5     mapping(address => uint256) public balances;
6
7     function depositFunds() public payable {
8         balances[msg.sender] += msg.value;
9     }
10
11    function withdrawFunds (uint256 _weiToWithdraw) public {
12        require(balances[msg.sender] >= _weiToWithdraw);
13        require(_weiToWithdraw <= withdrawalLimit);
14        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
15        require(msg.sender.call.value(_weiToWithdraw)());
16        balances[msg.sender] -= _weiToWithdraw;
17        lastWithdrawTime[msg.sender] = now;
18    }
19
20 }
```

```
1 contract Attack {
2
3  EtherStore public etherStore;
4  constructor(address _etherStoreAddress) {
5        etherStore = EtherStore(_etherStoreAddress);
6  }
7  function pwnEtherStore() public payable {
8        require(msg.value >= 1 ether);
9        etherStore.depositFunds.value(1 ether) ();
10        etherStore.withdrawFunds(1 ether);
11  }
12  function collectEther() public {
13        msg.sender.transfer(this.balance);
14  }
15  function () payable {
16        if (etherStore.balance > 1 ether) {
17            etherStore.withdrawFunds(1 ether);
18        }
19  }
20 }
```

Figure 2: Reentrancy vulnerability example.

```
1 contract Roulette {
2     uint public pastBlockTime;
3
4     // initially contract
5     constructor() {}
6
7     // receive function
8     receive() external payable {}
9
10     // fallback function used to make a bet
11     fallback() external payable {
12         require(msg.value == 1 ether); // must send 1 ether to play
13         // only 1 transaction per block
14         require(block.timestamp != pastBlockTime);
15         pastBlockTime = block.timestamp;
16         if(block.timestamp % 15 == 0) { // winner
17             payable(msg.sender).transfer(address(this).balance);
18         }
19     }
20 }
```

Figure 3: Timestamp dependency vulnerability example.

balance in the vulnerable contract after the transfer operation of call. Value (), then when the attacker calls the transfer function call, value () will trigger the rewritten fallback function of the attacker's contract. The transfer function can be called again to continuously recursively transfer money from the vulnerable contract to the attacker contract. Figure 2 contains an EtherStore contract and an attack contract. The EtherStore contract implements the Ether vault function, which allows users to withdraw one Ether coin per week. Lines 13 and 14 of the EtherStore contract code judge the withdrawal amount and the withdrawal interval, so that a withdrawal can only be performed successfully if the requested withdrawal amount is less than 1 ether and no withdrawals have been made in the last week. Line 15 of the code sends the requested ether to the user via the call.value () function. The attacker launches an attack on the EtherStore contract through the Attack contract and calls the EtherStore contract's withdrawFunds () function to withdraw 1 ether, at which time the requirements of the withdrawal amount and the withdrawal interval are met and the attack contract receives 1 ether from the EtherStore contract and executes the fallback function. In the fallback function, the withdrawFunds () function is called again to reenter the EtherStore contract. When the withdrawFunds () function is called for the second time, lines 16 and 17 of the code have not been executed, and the balance and withdrawal time still meet the contract requirements. Therefore, the attacker can continue to withdraw ether, thus allowing all ether to be withdrawn from the EtherStore contract in a single transaction.

2.3. Timestamp Dependence. Timestamp dependency means that the execution of the smart contract depends on the timestamp of the current block. With the different timestamps, the execution results of the contract also vary. Vulnerabilities arise when blockchain timestamps are used as seeds to generate random numbers or as various time-dependent state change conditional statements to perform certain critical operations [23]. Some variables exist in the block header, including BLOCKHASH, TIMESTAMP, NUMBER, GASLIMIT, and COINBASE, so in principle, they can be influenced by miners. Miners have the right to set block timestamps within a 900 second offset. If the timestamp of the new block is greater than the timestamp of the previous block, and the difference between the timestamps is less than 900 seconds, then the timestamp of the new block is legal. If the cryptocurrency is transmitted based on block variables, a malicious miner could change the timestamps of their blocks to exploit the vulnerability. Figure 3 shows a code example that contains a timestamp dependency vulnerability. The contract implements a lottery function where one transaction per block can be invested with 1 ether, and each player has a 1 in 15 chances of winning the contract

```
1 contract HashForEther {
2
3     function withdrawWinnings() {
4         // Winner if the last 8 hex characters of the address are 0.
5         require(uint32(msg.sender) == 0);
6         _sendWinnings();
7     }
8
9     function _sendWinnings() {
10        msg.sender.transfer(this.balance);
11    }
12 }
```

Figure 4: Implicit visibility vulnerability example.

balance according to the contract logic. The vulnerability occurs on line 15, where miners can adjust the block timestamp so that block timestamp takes 15 modulo 0 to win the block reward of ether locked in the contract.

2.4. Implicit Visibility. Functions in Solidity have visibility specifiers, which indicate how the function is called. Visibility determines whether a function can be called externally by the user, by other derived contracts, internally only, or externally only. Functions can be specified as external, public, internal, or private, and incorrect use of visibility specifiers can lead to a number of development vulnerabilities in smart contracts. The default visibility of functions is public, so external users can call functions without specifying any visibility. Vulnerabilities exist when a function is supposed to be private or can only be called within the contract itself, and the developer ignores the function's visibility specifier. The contract in Figure 4 contains an undeclared visibility vulnerability. HashForEther () is an address-guessing bounty game contract that allows users to call the withdrawinnings () function to get their bounty once they generate an Ethereum address with the last 8 hexadecimal characters being 0. However, withdrawinnings () and sendwinnings () do not specify that the function visibility is a public function, so any address can call this function to steal the bounty.

3. Related Work

Smart contract vulnerability detection is one of the fundamental issues of blockchain security. Early methods used static and dynamic analysis methods such as symbolic execution and fuzzing, which depended on expert-defined vulnerability rules [3]. Recent approaches utilize deep learning to build vulnerability detection models that automatically learn vulnerability features. This section will introduce the current research work from two aspects: traditional vulnerability detection methods based on static and dynamic analysis, such as symbol execution and fuzzy testing; and methods based on deep learning.

3.1. Traditional Vulnerability Detection Methods. Slither [24] is the first open source static analysis framework for the Solidity language. Slither converts Solidity smart contracts into an intermediate representation called Slither, which provides fine-grained information about smart contract code and can flexibly support many applications. Smartcheck [25] is an extensible static analysis tool that checks Solidity source code by converting it into an intermediate XML-based representation and then checking it against XPath schemas for comparison. Typical techniques for dynamic analysis include fuzzing and symbolic execution. Oyente [26], a dynamic symbolic execution detection tool, was the first tool for security analysis of smart contracts. ContractFuzzer [27] uses the fuzzing method to detect vulnerabilities in Ethereum smart contracts. It can generate fuzzing input according to the ABI specification of smart contracts and define test predictions for detecting security vulnerabilities. These methods all have the problem of low automation, and most of them need to rely on expert knowledge and human experience to extract fixed rules for vulnerabilities, which is inefficient and laborious. The static analysis method needs to manually set the matching rules according to the analysis efficiency of the matching rules, so it has a high false positive rate. Compared with static code analysis [28–30], dynamic analysis methods execute smart contracts in real blockchain systems, so dynamic analysis methods are more accurate. However, the debugging, analysis, and running of the target program in dynamic analysis requires a large number of personnel to participate, which has some disadvantages, such as slow speed, low efficiency, and the difficulty of carrying out large-scale testing.

3.2. Vulnerability Detection Method Based on Deep Learning. With the continuous development of artificial intelligence technology, there have been many studies applying machine learning, deep learning, and other technologies to the field of vulnerability mining to identify vulnerabilities in smart contracts. Gao [11] learned the structured code embedding of smart contracts based on deep learning methods, collected 52 known vulnerability contracts as a vulnerability database that contains 10 common vulnerabilities, and identified clone defects through the vulnerability database. Xu et al. [15] builds an abstract syntax tree for smart contracts, compares the ASTs of two smart contracts to obtain shared child nodes, and uses structural similarity to detect vulnerabilities. It builds AST of smart contracts from a manually injected vulnerability contract dataset, extracts subnodes shared with marked contracts, and obtains structural similarity through machine learning. However, there are various vulnerability modes in reality, and the contract construction template only by manually injecting vulnerabilities cannot contain all the vulnerability modes in reality. This method of constructing AST templates based on existing vulnerability contracts has poor scalability and can only detect existing vulnerability patterns. Zhuang et al. [31, 32] takes the opcode sequence as the input of the sequence learning model, and uses LSTM to learn the smart contract opcodes to obtain the vulnerability-related features. ContractWard [12] uses the n-gram algorithm to extract binary features from the opcodes of smart contracts to construct feature spaces, and uses machine learning methods to build models for vulnerability detection. Qian et al. [33] constructed a contract graph to represent the syntactic and semantic structure of smart contract functions, and used a graph

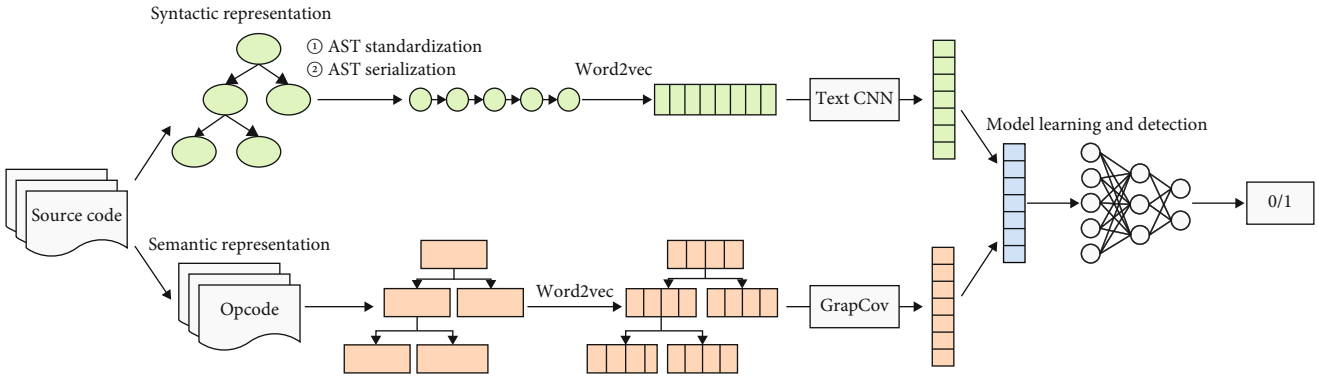


Figure 5: Overall framework.

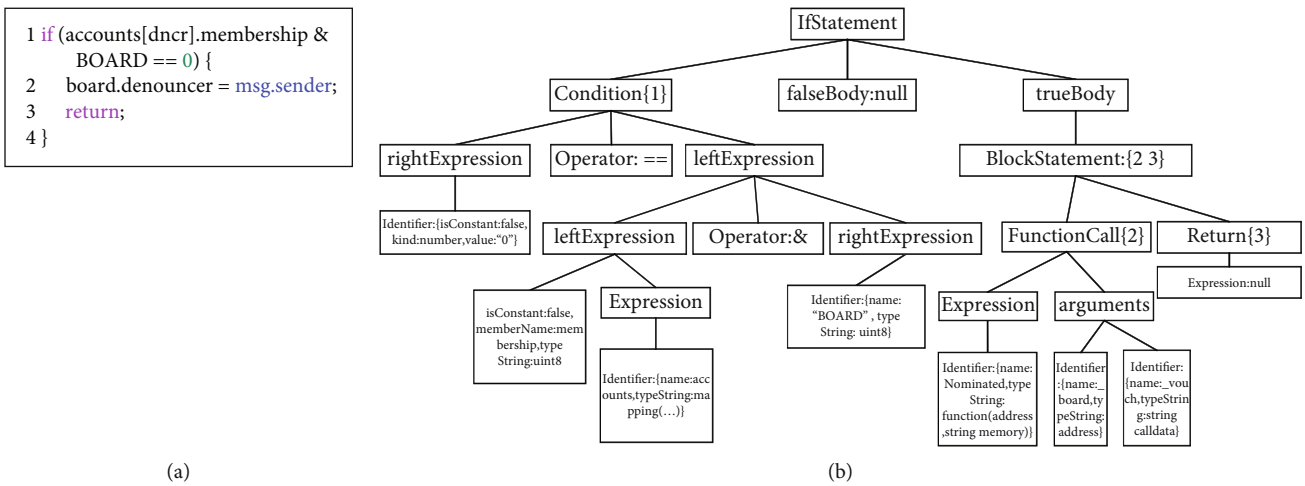


Figure 6: AST of solidity function (a) a code segment containing if statements. (b) CFG converted from code segment.

convolutional neural network to learn vulnerabilities from the contract graph. Compared with traditional automated vulnerability detection, deep learning-based vulnerability detection methods are more accurate and efficient, and their generalization ability is better. By comparing the current deep learning methods, it can be observed that the key to applying deep learning methods for vulnerability detection lies in the selection of a suitable representation for the source code. Selecting a suitable code representation and then building a feature extractor to extract the features of the source code can further improve the efficiency of vulnerability detection.

4. Method

In order to obtain the multiple features of the contract source code, the model extracts the syntactic features and semantic features from the source code, respectively, and fuses the syntactic and semantic features, using the fusion features to detect vulnerabilities. The overall framework of this paper is shown in Figure 5, which consists of three phases: syntax representation, semantic representation, and model learning and detection. Contract code embedding parses smart contracts into appropriate representation structures for model training. We use AST and CFG to represent smart contracts in two stages to abstract the feature information of source code vulnerabilities. In the syntax representation stage, the AST of the smart contract is obtained from the source code. The AST is normalized and serialized, and then transformed into a vector representation using word embedding. Finally, the syntax features are obtained using Text CNN. In the semantic representation stage, the source code is first compiled into bytecode, and the CFG of the smart contract is obtained from the bytecode. Then use word2vec to vectorize the graph node features and use MPNN to obtain the semantic and structural embeddings of the graph. In the model learning stage, combined with the syntactic vector and semantic vector obtained in the syntactic representation and semantic representation stages, the fully connected layer is used to learn the multifeature vector to train the detector. Finally, the vulnerability detection of smart contract source code is implemented using the trained detector.

4.1. Syntactic Representation. In the syntactic representation stage, we use the AST obtained by compiling the source code

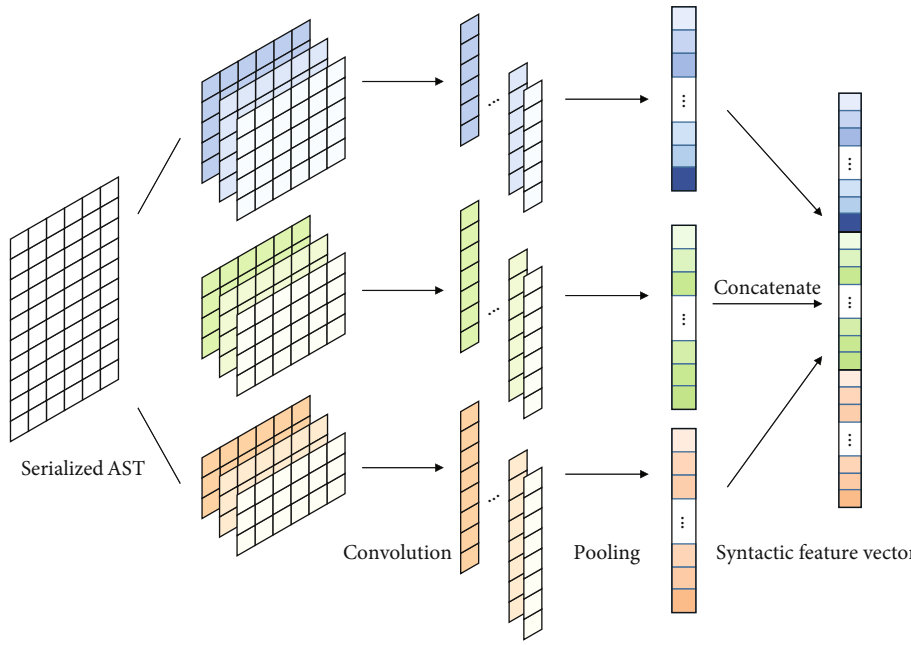


Figure 7: TextCNN model structure.

to represent the syntactic features of the code. AST is a tree representation of the syntax structure of a program's source code, with each node in the tree representing a structure in the source code. This syntax-based code representation can preserve the syntactic structure information of the source code, which represents the syntax and details of each statement, such as the number of functions defined, variable types, etc. Using AST for feature extraction can obtain more syntactic features than source code text.

(1) *Get the abstract syntax tree*. First, we obtain the AST of the smart contract. In this paper, we use py-solc-x (a third-party software package for Python) to build the AST of the smart contract. Py-solc-x converts the source code of a smart contract into a tree-structured AST. Figure 6(a) shows a code segment containing if statements. The transformed AST is shown in Figure 6(b), and each node contains information such as type, name, child node, and value. After obtaining the AST, we normalize, serialize, and vectorize the AST

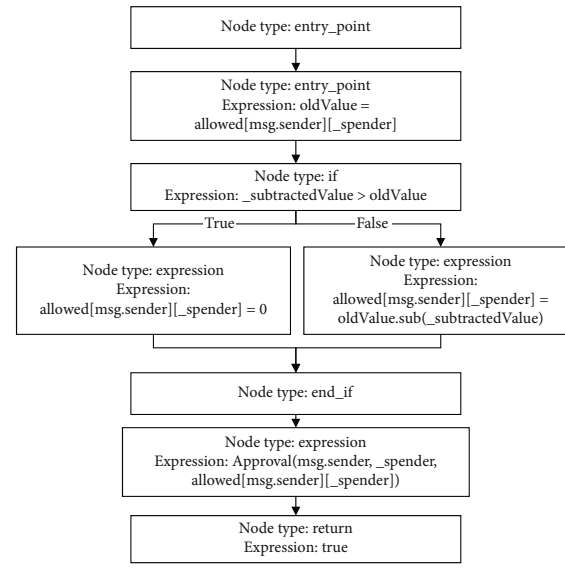
(2) *AST normalization*. The AST contains user-defined contract names, function names, variable names, and variable values. The user-defined names and values vary from contract to contract, which can affect the accuracy of vulnerability detection, leading to more false negatives. To eliminate this variability, we normalized the AST. According to the order in which the user-defined contract name, function name, and variable name appear in the contract, they are, respectively, represented as $(\text{contract } 1, \cdots, \text{contract } n)$, $(\text{function } 1, \cdots, \text{function } n)$, $(\text{var } 1, \cdots, \text{var } n)$. By normalizing, the model can be made to focus more on the contract structure and ignore this variability

(3) *AST serialization*. After normalization, we perform serialization operations on the AST. In order to extract the syntactic structure of the solidity contract, we traverse all nodes of the AST in depth-first order and transform the tree-structured AST into a serially structured code token. The AST constructed from source code is longer than the source code, and in order to be more conducive to model training, we only keep some key information during serialization, while ignoring some secondary information indicating code location and node id

(4) *Vectorization*. Smart contract source code is usually a text representation and cannot be directly used for training deep neural network models. After normalizing and serializing the AST, the AST needs to be represented in vector form as the input for training the model. Each code token is represented as a vector with the same dimension using word2vec [34], thereby representing the serialized AST in vector form. The source code of the contracts varies in length, so the transformed feature vectors have different lengths. We represent a text vector as a fixed-length numeric vector by truncation and padding operations. Pad with zeros when the vector length is less than the fixed length, and truncate when the vector length is greater than the fixed length

(5) *Syntax Feature Extraction*. Text CNN [35] is a text classification model proposed by Yoon Kim for the deformation of the input layer of CNN, which has many applications in the field of natural language processing [36, 37]. Text CNN makes the model

```
1  function decreaseApproval(address _spender, uint _subtractedValue)
2    public returns (bool) {
3   uint oldValue = allowed[msg.sender][_spender];
4    if (_subtractedValue > oldValue) {
5     allowed[msg.sender][_spender] = 0;
6    } else {
7     allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
8    }
9   Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
10    return true;
11 }
```



(a) (b)

Figure 8: CFG of solidity function (a) code for the decreaseApproval() function. (b) CFG of the function decreaseApproval().

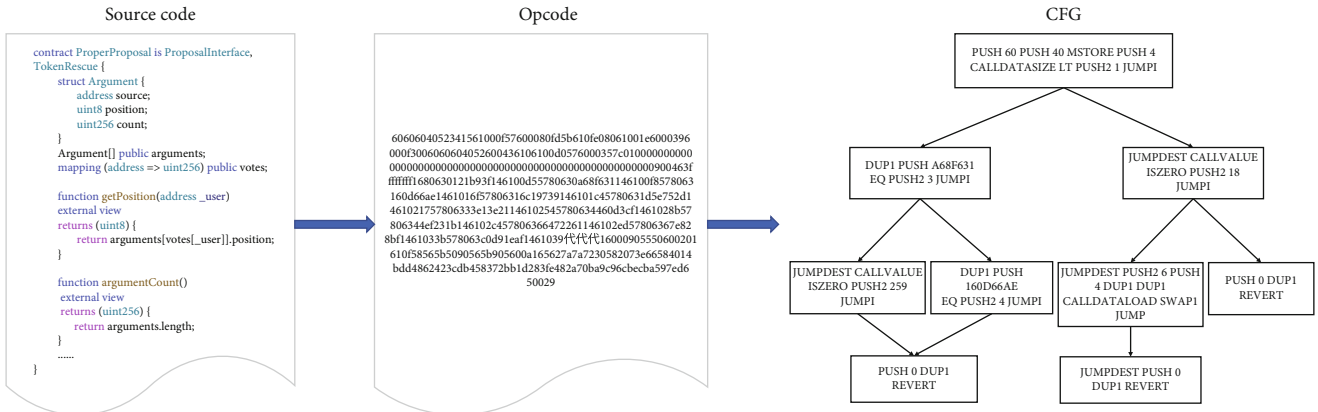


Figure 9: Control flow graph transformation process.

work well by introducing the trained word vectors, and can solve the problem of gradient disappearance of long texts. We use Text CNN to extract syntax features, and the model structure is shown in Figure 7. We define multiple 1D convolution kernels and perform convolution operations on the input separately. Convolution kernels of different widths can capture local features between different numbers of adjacent tokens. Adaptive average pooling is performed on all output channels, then all scalar convergence outputs are concatenated into vectors, and finally we obtain the syntactic representation of the smart contract

4.2. Semantic Representation. In the semantic representation stage, the CFG of each contract is obtained to extract the semantic features of the contract code. CFG is an abstract representation of a process or program, an abstract data structure used in a compiler that represents all the paths that a program will traverse during its execution. Each node in the CFG represents a basic block, that is, a piece of code without any jumps or jump targets, which is process-oriented and can reflect many information and execution flows of all basic blocks in a process. Figure 8(a) shows a contract function decreaseApproval() containing an if statement, and Figure 8(b) shows the CFG that the function is transformed into, displaying the execution flow of the function's statement. Through CFG, it is possible to traverse all the execution paths of a function, and discover the control dependencies and data dependencies that exist in the program. The data dependencies and control dependencies embodied by CFG are the missing information in the AST, which can capture more comprehensive semantic information, and thus the code semantic representation with CFG can complement the syntactic features extracted by AST. The semantic representation of code with CFG can supplement the syntactic features extracted by AST.

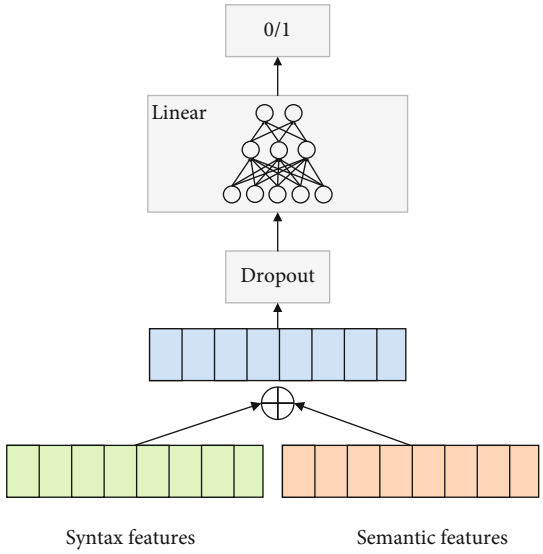


Figure 10: Feature fusion.

Table 1: Number of dataset samples.

Vulnerability type	No vulnerability	Vulnerability	Proportion
Implicit visibility	11677	11200	1.04 : 1
Integer overflow	14114	8763	1.61 : 1
Integer underflow	19236	3614	5.32 : 1
TimeDependency	22552	325	69.3 : 1
Reentrancy	22814	63	362 : 1

(1) *Get the control flow graph*. In the conversion of smart contracts to control flow graphs, the smart contracts are first compiled into opcodes, and then the opcodes are converted into control flow graphs by evm_cfg_builder. The converted CFG is shown in Figure 9, where each node is an independent block of code and the statements in the node are represented by opcodes

(2) *Block embedding*. In the processing of CFG, semantic information and structural information are obtained through two steps: block embedding and graph embedding. Each block in the CFG contains the statement information, and the block embedding through word2vec can preserve the opcode information of the source code. The statements in each block are represented as a fixed-length vector after block embedding, and the semantic vector is used as the feature of the graph node

(3) *Graph embedding*. After the node features of the CFG are obtained through block embedding, the CFG is represented by a graph neural network to obtain a graph embedding vector representing the semantics and structure of the entire CFG graph. Graph Convolutional Neural Networks [38] apply convolutional operations to graph data, aggregating node own features and domain node features to generate new node representations. In this paper, we use a Graph Convolutional Neural Network to update the CFG node representation and use the mean-pool readout layer to obtain the graph-embedding vector. Through block embedding and graph embedding, the final graph embedding representation preserves the overall structural information of the CFG, as well as the semantic information of each node's opcode

4.3. Model Learning and Vulnerability Detection. After obtaining the syntactic embedding and semantic embedding of the source code through the syntactic characterization and semantic characterization stages, the syntactic and semantic feature vectors are fused for model training. In this paper, we choose to perform vulnerability detection at the contract level. Since the AST obtained in the syntactic representation stage is at the file level, and the CFG obtained in the semantic representation is at the contract level, it is necessary to split the entire AST into multiple small contract ASTs before splicing the feature vectors. The fused feature vectors combined with source code syntax and semantic information can retain more feature information in the model learning stage. As shown in Figure 10, we fuse syntactic and semantic feature vectors as input to train a feedforward neural network for vulnerability detection. We take the obtained fused vector as the input to the model to automatically learn the potential code features from the fused feature vector. Finally, the learned vulnerability detection model is used in the vulnerability detection stage.

After training to obtain a vulnerability detection model, vulnerability detection can be performed on smart contracts. The process of the detection stage is the same as that of the learning stage in terms of data preprocessing and data representation. After the target program is vectorized by syntax representation and semantic representation, it is fed into the vulnerability inspection model obtained in the learning stage to obtain the prediction result. The method in this paper supports the detection of solidity source code and can directly detect the contract source code. When detecting smart contracts, the test contract is inputted into the embedding module of syntactic and semantic representation, and any given new smart contract is transformed into a vector representation by obtaining the AST and CFG of the contract and utilizing the learned embedding matrix. The smart contract is then detected using the vulnerability detection model obtained from the training to determine whether there is a vulnerability ("1") or no vulnerability ("0").

5. Experiments

In this section, we describe the vulnerability dataset used in this paper, analyze the ratio of vulnerable to nonvulnerable codes in the dataset, and conduct extensive experiments on the test set to evaluate the effectiveness of the model proposed in this paper. First, the performance of the detection model is measured using accuracy, precision, recall, and F1 metrics to illustrate the overall performance of our model. Secondly, the effectiveness of TextCNN for extracting

Table 2: Dataset experimental results.

	Methods	Accuracy	Precision	Recall	F1
ImplicitVisibility	Conkas	—	—	—	—
	Opcode	54.16	61.86	21.73	32.16
	Our model	**89.00**	**87.58**	**92.04**	**89.75**
IntegerOverflow	Conkas	56.63	24.64	87.51	38.45
	Opcode	83.28	81.71	85.75	83.68
	Our model	**95.58**	**94.08**	**97.79**	**95.90**
IntegerUnderflow	Conkas	87.01	63.09	72.76	67.58
	Opcode	72.70	71.61	75.20	75.48
	Our model	**96.42**	**92.86**	**88.51**	**90.63**
TimeDependency	Conkas	92.97	57.64	67.49	62.18
	Opcode	94.04	97.01	90.89	93.85
	Our model	**98.25**	**97.52**	**94.85**	**94.81**
Reentrancy	Conkas	83.02	49.80	70.22	58.27
	Opcode	87.51	81.48	**97.08**	**88.60**
	Our model	**98.64**	**90.91**	78.38	75.36

syntactic features is verified. Finally, the single-feature code representation and multifeature code representation are compared to verify the effectiveness of the multifeature vulnerability detection proposed in this paper.

5.1. Datasets and Data Preprocessing. This paper uses the Eth2Vec [39] dataset for model training. The Eth2Vec dataset contains 5,000 smart contract files of the real Ethereum environment, with a total of 22,879 contract functions. The labeling results of the five vulnerability types in the Eth2Vec dataset are shown in Table 1, and it can be observed that the number of positive and negative samples in the dataset is unbalanced. The number of positive and negative samples for Implicit Visibility and Integer Overflow vulnerabilities is relatively balanced, while the proportion of positive and negative samples for the other three types of vulnerabilities differs significantly. The datasets with reentrancy vulnerabilities and timestamp dependencies are the most unbalanced, with less than 400 positive examples. The unbalanced dataset will affect the training results of the model, which is not conducive to the model learning the features of classes with fewer samples. Therefore, we preprocess the dataset to alleviate the data imbalance problem. In this paper, we use a weighted random sampling method to process the dataset to balance the number of samples.

5.2. Results and Analysis. In this section, we illustrate the experimental results and performance comparisons of our model on smart contract vulnerability detection. For a dataset with few positive samples, the model may detect all samples as negative samples to obtain a higher accuracy, so we should also pay attention to the recall metric while focusing on the accuracy. The recall metric represents the ratio of predicted correct positive samples to true positive samples, reflecting the ability of the model to effectively detect vulnerabilities. We evaluate the effectiveness of the model with accuracy, precision, recall, and F1-score.

5.2.1. Overall Performance. In order to verify the performance of the model in this paper, we compared it with the current detection methods, and the experimental results are shown in Table 2. First, we compare it with traditional vulnerability detection methods. Conkas [40] is a smart contract static analysis tool that detects vulnerabilities using symbolic execution. Conkas can detect vulnerability types such as arithmetic, timestamp dependency, and reentrancy vulnerability. Second, we compare it with deep learning-based detection methods. Opcode [31] detects smart contract security vulnerabilities at the opcode level, so we first compile the contract into opcodes and then detect the opcodes. Compared with Conkas and Opcode, our model achieves better performance on all five detection tasks. In the reentrancy vulnerability detection, there are only 63 contracts with vulnerabilities, with a large difference in the number of positive and negative samples. Although balanced by weighted random sampling, there are still too few vulnerability features that can be learned, so the recall for reentrancy vulnerability detection is low compared to other vulnerability types. The recall rates of Integer Overflow and Integer Underflow vulnerabilities reached 97.79% and 88.51%, with performance improvements of 10.28% and 13.31%, respectively. The experimental results illustrate the effectiveness of our model.

5.2.2. Validity of Syntactic Representation. To verify the effectiveness of the syntactic representation module for AST syntactic feature extraction, we compared it with LSTM-AST and GRU-AST. We replace the syntactic feature extraction part of the model with LSTM and RNN, while the other parts of the model remain unchanged. LSTM-AST indicates that the feature extraction part of the syntactic representation is replaced by LSTM to extract the syntactic features of AST. GRU-AST indicates that the syntactic features of AST are extracted by GRU [41]. In the syntactic representation of the source code, we chose Text CNN for syntactic

Table 3: Syntax representation module experimental results.

	Methods	Accuracy	Precision	Recall	F1
ImplicitVisibility	LSTM-AST	85.50	84.82	88.06	86.41
	GRU-AST	86.24	85.32	86.46	86.95
	Our model	**89.00**	**87.58**	**92.04**	**89.75**
IntegerOverflow	LSTM-AST	94.17	93.52	95.58	94.54
	GRU-AST	93.92	92.28	95.05	94.16
	Our model	**95.58**	**94.08**	**97.79**	**95.90**
IntegerUnderflow	LSTM-AST	96.17	92.76	87.23	89.91
	GRU-AST	95.83	89.70	**88.94**	89.32
	Our model	**96.42**	**92.86**	88.51	**90.63**
TimeDependency	LSTM-AST	94.29	85.48	88.79	87.10
	GRU-AST	95.92	89.61	89.22	89.42
	Our model	**98.25**	**97.52**	**94.85**	**94.81**
Reentrancy	LSTM-AST	97.83	68.97	54.05	60.61
	GRU-AST	97.92	63.64	70.27	69.14
	Our model	**98.64**	**90.91**	**78.38**	**75.36**

Table 4: Comparison of experimental results of each module.

		Accuracy	Precision	Recall	F1
ImplicitVisibility	CFG	86.08	85.96	87.74	86.84
	AST	88.00	**92.76**	83.60	87.94
	AST + CFG	**89.00**	87.58	**92.04**	**89.75**
IntegerOverflow	CFG	92.92	91.03	96.06	93.48
	AST	94.08	93.37	95.58	94.47
	AST + CFG	**95.58**	**94.08**	**97.79**	**95.90**
IntegerUnderflow	CFG	95.25	91.59	83.40	87.31
	AST	94.66	90.91	80.85	85.59
	AST + CFG	**96.42**	**92.86**	**88.51**	**90.63**
TimeDependency	CFG	95.42	89.33	86.64	87.96
	AST	98.25	96.48	92.24	**95.42**
	AST + CFG	**98.25**	**97.52**	**94.85**	94.81
Reentrancy	CFG	97.33	58.06	48.65	52.94
	AST	98.58	76.32	73.68	**77.33**
	AST + CFG	**98.64**	**90.91**	**78.38**	75.36

feature extraction, considering that the AST sequence transformed into the source code is long and the word order information is not obvious. The experimental results are shown in Table 3. The syntactic representation module achieves optimality in all four metrics for the four vulnerability types of detection of ImplicitVisibility, IntegerOverflow, IntegerUnderflow, TimeDependency, and Reentrancy. The syntax characterization module improves precision and recall by 2%-7% on ImplicitVisibility, IntegerOverflow, and IntegerUnderflow vulnerability detection compared to the replaced model. The precision of TimeDependency and Reentrancy vulnerability detection has improved by 8% and 22%, and the recall rate has improved by 5% and 8%. In IntegerUnderflow vulnerability detection, although the syntactic representation module has a slightly lower recall than GRU-AST, but the performance of the other three metrics is improved, and it obtains a more balanced performance. The experimental results show that the syntactic representation module can effectively extract the syntactic features of the source code. Compared with the sequence model, its performance is more balanced and stable in syntactic feature extraction for AST. We observe that the sequence model focuses more on the sequence order information, but the AST sequence does not have significant

temporal order information. TextCNN defines different filters to extract source code features from different perspectives, insensitive to word order information, and more suitable for AST sequences. Therefore, the syntactic features of the source code can be obtained more efficiently using TextCNN.

5.2.3. Effectiveness of Fusion Code Representation. We compare the single code representation with the multifeature code representation to verify the validity of each module and the validity of the fused code representation. To verify the validity of the syntactic representation module, we only use the AST representation of the source code for vulnerability detection. To verify the validity of the semantic representation module, only the semantic representation module is used for feature extraction of CFG. The experimental results are shown in Table 4. By comparing the four metrics, the vulnerability detection performance using AST is slightly better than CFG. This is because a part of the feature information is lost in the semantic representation module when representing the semantic information of each block as a fixed-length feature vector. In addition, the multifeature code representation improves the most in the recall metric, and the model has higher performance and accuracy in correctly identifying vulnerable codes. The highest accuracy and recall are achieved by fusion learning to use AST and CFG, which validates that our proposed syntactic representation and semantic representation modules can capture more features of source code for vulnerability detection.

6. Conclusions

In this paper, we propose a new approach for smart contract vulnerability detection, which characterizes the code by fusion learning of syntactic and semantic information. The syntactic and semantic features in the source code are automatically learned through the syntactic representation and semantic representation modules, and the syntactic-semantic fusion vectors are used to detect smart contract vulnerabilities. We conducted experiments on the smart contract dataset in Ethereum, and the results show that our method can accurately identify multiple types of vulnerabilities, with significant improvements in precision and recall compared to existing methods. Our model detects against contract source code without defining complex vulnerability patterns, enabling effective and automated vulnerability detection. In the following work, we will explore more effective vulnerability detection models that can localize the location of the vulnerable code while accurately detecting the vulnerability.

Data Availability

Previously reported smart contract source code data were used to support this study and are available at https://github.com/fseclab-osaka/eth2vec. These prior studies and datasets are cited at relevant places within the text as references Eth2Vec.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This research was funded by the National Natural Science Foundation of China (41871316), the Scientific and technological project in Henan Province (212102210414 and 22B520003), and the Foundation of University Young Key Teacher of Henan Province (Grant No. 2019GGJS040 and 2020GGJS027).

References

[1] N. Szabo, "Smart contracts: building blocks for digital markets," 1996, https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[2] X. B. Wang, X. Y. Yang, X. F. Shu, and L. Zhao, "Formal verification of smart contracts based on MSVL," *Journal of Software*, vol. 6, no. 32, pp. 1849–1866, 2021.

[3] Z. H. A. N. G. Ying-li, M. A. Jia-li, L. I. U. Zi-ang, L. I. U. Xin, and Z. H. O. U. Rui, "Overview of vulnerability detection methods for Ethereum solidity smart contracts," vol. 49, Tech. Rep. 3, Computer Science, 2022.

[4] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, "Efficient vulnerability detection based on abstract syntax tree and deep learning," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 722–727, Toronto, ON, Canada, 2020.

[5] L. Lv, Z. Wu, J. Zhang, L. Zhang, Z. Tan, and Z. Tian, "A VMD and LSTM based hybrid model of load forecasting for power grid security," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 9, pp. 6474–6482, 2022.

[6] S. D. Okegbile, J. Cai, C. Yi, and D. Niyato, "Human digital twin for personalized healthcare: vision, architecture and future directions," *IEEE Network*, pp. 1–7, 2022.

[7] D. Cao, J. Huang, X. Zhang, and X. Liu, "FTCLNet: convolutional LSTM with fourier transform for vulnerability detection," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 539–546, Guangzhou, China, 2020.

[8] C. Yi, J. Cai, and Z. Su, "A multi-user mobile computation offloading and transmission scheduling mechanism for delay-sensitive applications," *IEEE Transactions on Mobile Computing*, vol. 19, no. 1, pp. 29–43, 2020.

[9] K. B. Kim and J. Lee, "Automated generation of test cases for smart contract security analyzers," *IEEE Access*, vol. 8, pp. 209377–209392, 2020.

[10] J. Qiu, Z. Tian, C. Du, Q. Zuo, S. Su, and B. Fang, "A survey on access control in the age of internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4682–4696, 2020.

[11] Z. Gao, "When deep learning meets smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1400–1402, Virtual Event, Australia, 2020.

[12] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: automated vulnerability detection models for Ethereum

smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2021.

[13] L. Zhang, Z. Huang, W. Liu, Z. Guo, and Z. Zhang, "Weather radar echo prediction method based on convolution neural network and long short-term memory networks for sustainable e-agriculture," *Journal of Cleaner Production*, vol. 298, article 126776, 2021.

[14] L. Lv, Z. Wu, L. Zhang, B. B. Gupta, and Z. Tian, "An edge-AI based forecasting approach for improving smart microgrid efficiency," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 11, pp. 7946–7954, 2022.

[15] Y. Xu, G. Hu, L. You, C. Cao, and A. Derhab, "A novel machine learning-based analysis model for smart contract vulnerability," *Security and Communication Networks*, vol. 2021, Article ID 5798033, 12 pages, 2021.

[16] L. Zhang, Y. Huo, Q. Ge, Y. Ma, Q. Liu, and W. Ouyang, "A privacy protection scheme for IoT big data based on time and frequency limitation," *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 5545648, 10 pages, 2021.

[17] C. Qiao, K. Brown, F. Zhang, and Z. Tian, "Federated adaptive asynchronous clustering algorithm for wireless mesh networks," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[18] Y. Sun, Z. Tian, M. Li, S. Su, X. Du, and M. Guizani, "Honeypot identification in softwarized industrial cyber-physical systems," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5542–5551, 2021.

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

[20] C. Yi, J. Cai, T. Zhang, K. Zhu, B. Chen, and Q. Wu, "Workload re-allocation for edge computing with server collaboration: a cooperative queueing game approach," *IEEE Transactions on Mobile Computing*, p. 1, 2021.

[21] Y. Sun, Z. Tian, M. Li, C. Zhu, and N. Guizani, "Automated attack and defense framework toward 5G security," *IEEE Network*, vol. 34, no. 5, pp. 247–253, 2020.

[22] L. Zhang, C. Xu, Y. Gao, Y. Han, X. du, and Z. Tian, "Improved Dota2 lineup recommendation model based on a bidirectional LSTM," *Tsinghua Science and Technology*, vol. 25, no. 6, pp. 712–720, 2020.

[23] L. Lv, J. Chen, Z. Zhang, B. Wang, and L. Zhang, "A numerical solution of a class of periodic coupled matrix equations," *Journal of the Franklin Institute*, vol. 358, no. 3, pp. 2039–2059, 2021.

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

[25] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, Vienna, Austria: Association for Computing Machinery, 2016.

[26] B. Jiang, Y. Liu, and W. K. Chan, "Contract fuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering: Association for Computing Machinery*, pp. 259–269, Montpellier, France, 2018.

[27] W. J.-W. Tann, X. J. Han, S. Sengupta, and Y. J. A. Ong, "Towards safer smart contracts: a sequence learning approach to detecting vulnerabilities," 2018, https://arxiv.org/abs/1811.06632.

[28] C. Qiao, J. Qiu, Z. Tan, G. Min, A. Y. Zomaya, and Z. Tian, "Evaluation mechanism for decentralized collaborative pattern learning in heterogeneous vehicular networks," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–10, 2022.

[29] L. Zhang, S. Tang, and L. Lv, "An finite iterative algorithm for sloving periodic Sylvester bimatrix equations," *Journal of the Franklin Institute*, vol. 357, no. 15, pp. 10757–10772, 2020.

[30] C. Yi, J. Cai, K. Zhu, and R. Wang, "A queueing game based management framework for fog computing with strategic computing speed control," *IEEE Transactions on Mobile Computing*, vol. 21, no. 5, pp. 1537–1551, 2022.

[31] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural networks," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence; Yokohama*, Yokohama, Japan, 2021.

[32] L. Lv, S. Tang, and L. Zhang, "Parametric solutions to generalized periodic Sylvester bimatrix equations," *Journal of the Franklin Institute*, vol. 357, no. 6, pp. 3601–3621, 2020.

[33] P. Qian, Z. G. Liu, Q. M. He, B. T. Huang, D. Z. Tian, and X. Wang, "Smart contract vulnerability detection technique: a survey," *Journal of Software*, vol. 33, no. 8, pp. 3059–3085, 2022.

[34] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems-Volume 2*, pp. 3111–3119, Lake Tahoe, Nevada: Curran Associates Inc, 2013.

[35] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751, Doha, Qatar, Association for Computational Linguistics, 2014.

[36] C. Yi, S. Huang, and J. Cai, "Joint resource allocation for device-to-device communication assisted fog computing," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 1076–1091, 2021.

[37] L. Lv, J. Chen, L. Zhang, and F. Zhang, "Gradient-based neural networks for solving periodic Sylvester matrix equations," *Journal of the Franklin Institute*, vol. 359, no. 18, pp. 10849–10866, 2022.

[38] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017, https://arxiv.org/abs/1609.02907.

[39] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: learning contract-wide code representations for vulnerability detection on Ethereum smart contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pp. 47–59, Virtual Event, Hong Kong, 2021.

[40] N. Veloso, Ed., "Conkas: a modular and static analysis tool for Ethereum bytecode," 2021, https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997262417/94080-Nuno-Veloso_resumo.pdf.

[41] K. Cho, B. van Merrienboer, C. Gulcehre et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar. Association for Computational Linguistics, 2014.