International Journal of Data Warehousing and Mining
Volume 19 • Issue 2

An Efficient Code-Embedding-Based Vulnerability Detection Model for Ethereum Smart Contracts

Zhigang Xu, Hubei University of Technology, China
Xingxing Chen, Hubei University of Technology, China
Xinhua Dong, Hubei University of Technology, China*
Hongmu Han, Hubei University of Technology, China
Zhongzhen Yan, Hubei University of Technology, China
Kangze Ye, Hubei University of Technology, China
Chaojun Li, Hubei University of Technology, China
Zhiqiang Zheng, Narcotics Control Bureau of Department of Public Security of Guangdong Province, China
Haitao Wang, Narcotics Control Bureau of Department of Public Security of Guangdong Province, China
Jiaxi Zhang, Narcotics Control Bureau of Department of Public Security of Guangdong Province, China

ABSTRACT

Efficient and convenient vulnerability detection for smart contracts is a key issue in the field of smart contracts. The earlier vulnerability detection for smart contracts mainly relies on static symbol analysis, which has high accuracy but low efficiency and is prone to path explosion. In this paper, the authors propose a static method for vulnerability detection based on deep learning. It first disassembles Ethereum smart contracts into opcode sequences and then converts the vulnerability detection problem into a natural language text classification problem. The word vector method is employed to map each opcode to a uniform vector space, and the opcode sequence matrix is trained by the TextCNN method to detect vulnerabilities. Furthermore, a code obfuscation method is given to enhance and balance the dataset, while three different opcode sequence generation methods are proposed to construct features. The experimental results verify that the average prediction accuracy of each smart contract exceeds 96%, and the average detection time is less than 0.1 s.

KEYWORDS

Disassembly, Embedding, Ethereum, Machine Learning, Natural Language Processing, Smart Contracts, Static Method, Vulnerability Detection

INTRODUCTION

Blockchain is a chained storage structure (Li et al., 2022) that guarantees the security of the system through cryptography and other technologies, the consistency of transactions through consensus

DOI: 10.4018/IJDWM.320473 *Corresponding Author

algorithms (Fu et al., 2021), and the distributed storage of data through P2P networks. Due to its advantages in decentralization and traceability and immutability, blockchain technology has been widely used in industries such as information sharing (Park et al., 2021), product traceability (Wang et al., 2020), copyright protection (Liu et al., 2021), supply chain (Pournader et al., 2020), and finance (Kowalski et al., 2021). Currently, the development of blockchain is divided into three stages: blockchain 1.0, blockchain 2.0, and blockchain 3.0. Blockchain 1.0 is featured by programmable currency, represented by bitcoin Nakamoto (2008), with which, in 2017, saw its value soar by 1,900% (Holub & Johnson, 2019), to an extent where the price of a single Bitcoin reached $60,000 by 2021. Blockchain 2.0 is characterized by a programmable blockchain represented by Ethereum, of which the core is smart contracts. Ethereum is an open source public blockchain platform with smart contracts, and most smart contracts across the network currently run on the Ethereum platform. Blockchain 3.0 is a programmable community that goes beyond cryptocurrency and finance and is dedicated to applying blockchain to all aspects of life, providing decentralized solutions for various industries and moving toward the era of the smart Internet of Things. At present, the research on blockchain technology is in the intermediate stage from the blockchain 2.0 era to the blockchain 3.0 era. The research in this paper focuses on Ethereum smart contracts.

As an application platform for blockchain technology, Ethereum provides the largest execution platform for the operation of smart contracts. As a blockchain-based platform, Ethereum not only has basic cryptocurrency functions but also provides anonymous voting, transaction storage, product traceability, and other services. An Ethereum smart contract is an application running on an Ethereum virtual machine (EVM) in the form of EVM bytecode. Due to the open and transparent nature of Ethereum's own mechanisms, however, the EVM bytecode of a smart contract deployed on Ethereum can be accessed and analyzed by anyone. Although smart contracts are provided with a safe and secure execution environment by the blockchain, they may still face considerable security vulnerabilities in the development process due to the design mechanism of the Ethereum virtual machine, the problems carried by the characteristics of the blockchain, and the uneven code level of smart contract developers. Damage could be caused if an attacker finds a vulnerability by analyzing the EVM bytecode and launches an attack against the vulnerable smart contract, especially if the smart contract is used to handling asset-related business. The damage would be incalculable not only in terms of loss of assets but also in terms of challenge to the credibility and security of the platform. For example, in June 2016, the DAO incident occurred (Mehar et al., 2019), which directly led to a hard fall of Ethereum, where attackers exploited vulnerabilities in the DAO smart contract and stole 30% of the DAO's Ether in six hours, amounting to 12 million coins, with a market value of approximately $60 million. In July 2017, the Ethereum wallet parity was exposed to a major security flaw (Praitheeshan et al., 2019) in its multi-signature contract wallet.sol, which led to the theft of 150,000 Ether. Since smart contracts deployed on the blockchain cannot be modified, it is critical to verify smart contracts for crucial security vulnerabilities before deploying them to ensure that they are as secure as possible.

Figure 1 shows an example of a smart contract written in the Solidity language. The smart contract shown in Figure 1 has an integer overflow vulnerability. In lines 5 and 6, if a variable of type uint256 reaches its maximum value (2^256-1), the outcome will turn into 0 when a value greater than 0 is added. This is an integer overflow vulnerability. In lines 10 and 11, if a variable of type uint256 reaches its minimum value of 0, and a value greater than 0 is subtracted, its value will become 2^256-1 (the maximum value of type uint256). The current mainstream vulnerability detection tools, such as Oyente and Mythril, are inefficient and cannot detect new vulnerabilities in a timely manner.

This paper proposes a tool called Con2Vec (Contract to Vector), a static analysis tool based on machine learning, which learns smart contract code features through EVM bytecode and opcodes to detect smart contract vulnerabilities. It has a high throughput and accuracy rate and is user-friendly for those without any expertise concerning smart contract vulnerabilities. This paper builds Con2Vec in four parts starting with data cleaning and data augmentation to balance and augment the public dataset. Then the smart contract source code is compiled and decompiled into opcodes. The extracted

Figure 1.
Example of Smart Contract With IntegerFlow

```
1   pragma solidity ^0.4.25;
2   contract IntegerFlow{
3       //overflow
4       function add_overflow() returns (uint256 _overflow){
5           uint256 max = 2**256 - 1;
6           return max+1;
7       }
8       //underflow
9       function sub_underflow() returns (uint256 _overflow){
10          uint256 min = 0;
11          return min-1;
12      }
13  }
```

opcodes are then mapped to vector space by training the SkipGram model so that each form of opcode has a unique vector representation. Finally, the TextCNN model is used to detect vulnerabilities in the contracts.

This paper contributes as follows:

1) This paper proposes a method for code obfuscation of Ethereum smart contracts, which enhances the dataset and resists the interference of redundant codes during detection by inserting random useless codes into the smart contracts. Experiments show that the method can effectively generate enough obfuscated samples to enhance the detection model's resistance to redundant codes.
2) Through in-depth analysis and experiments on opcodes, three opcode sequence generation methods are proposed, and the samples generated by the three methods are trained and detected. It is found that the detection effect based on Opcode sequence is better than the other two. The larger the sample size is, the better the model can extract features based on Opcode sequence, and the higher the detection accuracy is.
3) This paper provides a vulnerability detection method where Con2Vec maps the codes to a unified vector space by porting the SkipGram model of natural language processing from the natural language level to the smart contract code level. By embedding the smart contract codes through a neural network, the model is able to extract information about the hidden features between the codes, as each code has its own code semantics. By learning and training the semantic features of the smart contract code, the model can predict whether vulnerabilities exist and what kinds of vulnerabilities exist in the input smart contract. The experiments show that the vulnerability detection rate of this method is significantly higher than that of the method in SoliAudit, and the average prediction accuracy of the system is over 96%.

The remainder of this paper is organized as follows: related work is summarized in the following section. Machine learning, natural language processing, Ethereum smart contracts and their associated vulnerabilities are introduced, followed by the description of the data imbalance problems and models. Experiments and data are then presented. The results are provided, followed by a discussion and final conclusion.

RELATED WORK

As the Ethereum blockchain is used in different areas, the security of Ethereum smart contracts is receiving increasing public attention. Many security analysis tools for detecting vulnerabilities in

Ethereum smart contracts have emerged in the industry. Vulnerability detection for smart contracts is generally divided into two types: static detection (Durieux et al., 2020; Tikhomirov et al., 2018; W. Wang et al., 2020) and dynamic detection (Grieco et al., 2020; Jiang et al., 2018; Wüstholz & Christakis, 2020). Static detection refers to the analysis of smart contract source code or bytecode. First, the malicious code is abstracted and its features are extracted. Then, the analysis is conducted through matching or similarity calculation methods. This method enjoys a high detection efficiency and a low omission rate, but it also leads to a high false alarm rate due to the variability of functions and features in smart contracts. Dynamic detection means that the smart contract code takes a pre run on the EVM, and the vulnerability is detected during the run. The method has high detection accuracy, but as it has to run through an EVM, its detection efficiency is low. At the same time, generating efficient test cases is difficult.

SmartCheck (Tikhomirov et al., 2018) is a vulnerability mining tool based on static detection for Ethereum smart contracts, mainly used to detect reentrancy, timestamp dependency, denial of service, and money locking vulnerabilities in smart contracts. First, the Solidity source code is analyzed syntactically and lexically. Then, the results of the analyzed syntactic tree are abstracted in the form of XML. Finally, the smart contract vulnerability is detected by XPath (Chen et al., 2021). ContractWard (W. Wang et al., 2020) is a vulnerability detection model based on machine learning. This model adopts the N-Gram method to model Ethereum smart contract opcodes and trains and predicts vulnerability samples through the XGBoost method for detecting six types of vulnerabilities including integer overflow, integer underflow, TOD, CallDepth, timestamp dependency, and reentrancy. Zhuang et al. (2020) proposed a smart contract vulnerability detection method based on graph neural networks. The method represented the syntax and semantic structure of smart contracts by constructing a contract graph, achieving good experimental results. ESCORT (Lutz et al., 2021) is an Ethereum smart contract vulnerability detection framework based on deep neural networks (DNN). This method applies transfer learning to smart contract vulnerability detection and supports the detection of unknown security vulnerabilities. Eth2Vec (Ashizawa et al., 2021) analyzes the bytecode by a specific extractor and uses a neural network to train the bytecode. The method detects smart contract vulnerabilities by comparing the code similarity between the target EVM bytecode and the trained EVM bytecode.

ContractFuzzer (Jiang et al., 2018) is a fuzzing framework for detecting vulnerabilities in smart contracts on the Ethereum platform. It generates inputs that fit the smart contract invocation syntax by analyzing the ABI interface of the smart contract and detects smart contract vulnerabilities by defining different test cases for distinct types of vulnerabilities. The tool can detect seven types of vulnerabilities including insufficient gas, exception passing, timestamp dependency, code injection, reentrancy, asset freezing, and transaction sequence dependency. Dynamit (Eshghie et al., 2021) is a dynamic monitoring framework based on machine learning. The method relies only on the transaction metadata and balance data of the blockchain system, extracts features from the transactions, and then analyzes the features through machine learning methods to detect reentrancy vulnerabilities of smart contracts. HFContractFuzzer (Ding et al., 2021) is a Hyperledger Fabric smart contract vulnerability detection tool based on fuzzy detection. The method combines the Golang fuzzing tool named go-fuzz with fabric smart contracts and detects three types of vulnerabilities including type conversion errors, logic loopholes, and integer overflow, verifying the feasibility of applying fuzzy techniques to HF smart contract vulnerability detection. Torres et al. (2021) proposed the first hybrid fuzzer for smart contracts, ConFuzzius, which uses evolutionary fuzzing to execute the shallow part of a smart contract and generates inputs that satisfy complex conditions by constraint solving, thus preventing evolutionary fuzzing from exploring the deep part and reducing low code coverage and false positives.

There are still problems that need to be solved for the vulnerability detection of Ethereum smart contracts. First, the current theoretical analysis for smart contract vulnerability detection is predominant, and most of the studies do not provide landing solutions that can be learned from. Second, most of the existing static detection methods extract and model features at the code level,

lacking feature analysis of the details between codes, which wastes the detailed information implied in the code files and reduces the efficiency of vulnerability detection and vulnerability coverage.

Due to the low disclosure rate of existing Ethereum smart contract code, which is mostly compiled EVM bytecode, the aim is to develop a static detection tool to analyze the source code or EVM bytecode directly and then carry out feature extraction and modeling mapping on the results obtained from the analysis to detect various vulnerabilities in smart contracts by analyzing EVM bytecode files. With this tool, only the source code or EVM bytecode of the smart contract is provided as input to the model, and the vulnerabilities are identified without executing the smart contract. Static detection is therefore often used as the first line of defense to avoid deploying a smart contract containing vulnerabilities on the blockchain. However, the accuracy of traditional static analysis tools for vulnerability detection is low because most of them are rule-based or based on code-level feature modeling, lacking feature extraction for detailed aspects of the code. Some static analysis approaches are executed by extracting the symbols of the control flow graph from the target code (Chinen et al., 2020; Torres et al., 2018; Weiss & Schütte, 2019), the generation of which requires traversing all states and is time-consuming.

To address the inherent shortcomings of traditional static analysis, many scholars have used machine learning methods to learn the features of the code as a way to model and infer whether the smart contract is vulnerable. The methods based on traditional machine learning, however, cannot extract the hidden features between smart contract codes, whereas the variability between code structures can seriously affect the analysis results. For instance, two methods with the same semantics but different code structures may turn out different results, and developers may deliberately add redundant code to evade detection by the vulnerability software, impacting the detection model. The existing smart contract vulnerability detection tools, based on static analysis, do not identify the hidden information between smart contract codes and are not robust enough to analyze different vulnerabilities. The Con2Vec vulnerability detection model, proposed in this paper, provides faster detection and higher scalability than the above tools. For this model, no prior knowledge of smart contract vulnerability mining is needed, and only a sufficient number of vulnerability samples need to be put into the model for training to obtain the detection model. If new vulnerability samples are subsequently obtained, the model can be enhanced by adding the corresponding samples directly to the model for incremental training.

BACKGROUND

This section provides a brief introduction to Ethereum smart contracts and their associated vulnerabilities, as well as concepts related to machine learning and natural language processing for detection schemes.

Ethereum Smart Contracts and Their Associated Vulnerabilities

Ethereum smart contracts run on the Ethereum virtual machine. There are two types of accounts in Ethereum: an externally owned account and a contract account. The externally owned accounts are controlled by keys, while the contract accounts are controlled by smart contract codes. Anyone can develop smart contracts on the Ethereum blockchain, but only contract accounts can own these smart contract codes. However, the externally owned account has a key through which the externally owned account can access the corresponding smart contracts. The contract account cannot start and run its own smart contract by itself and must initiate transactions to the contract account through an externally owned account, thus initiating the execution of the smart contract code in the contract account. An Ethereum smart contract is an immutable computer program deployed on the Ethereum blockchain that defines rules to be followed by all peers.

Smart contracts have three properties: immutability, transparency, and certainty. Immutability indicates that smart contract codes can be considered trustworthy because they cannot be modified

and removed once deployed to the blockchain. Transparency indicates that anyone with access to the blockchain can read the smart contracts on the blockchain. Certainty indicates that the same smart contract code will produce the same result regardless of who invokes it. Once a smart contract has been deployed to the blockchain, the contract will be executed automatically and will be triggered to handle the relevant functions when the conditions in the contract are met. Smart contracts allow anonymous participants to enter into binding agreements where each participant has full knowledge of the transaction, and value can be transferred between accounts or placed in a third party escrow within the smart contract.

To incentivize the execution of contract functions, Ethereum relies on gas paid in Ether to drive the movement of smart contracts. The amount of gas spent on a transaction is related to the complexity of the computation. According to the Ethereum protocol, a fee is charged for each computational step performed in a contract or transaction, thus preventing malicious attacks and abuse on the Ethereum network. Every transaction must include a gas limit as well as a fee willing to pay for gas. Miners can choose whether to pack the transaction and charge a fee. The transaction is executed if the total amount of gas in the computation step (gas used, including the original message and any sub messages that may be triggered) is less than or equal to the gas limit. All changes are rolled back if the total amount of gas exceeds the gas limit, unless the transaction is still valid and the miner accepts the cost. Any excess gas not used in the execution of the transaction is returned to the transaction initiator in Ether without fear of overspending, as only the cost of the consumed gas is paid during execution, meaning that it is useful and safe to send transactions above the estimated value of the gas limit.

Ethereum smart contracts are usually written in high-level languages such as Solidity, and the smart contract source code is compiled into a bytecode file in an EVM, which is a global single instance with unique results. It runs like a single instance computer between all peer nodes in the blockchain network, with each peer running a local copy of the EVM, thus verifying that the contract functions execution, and the transactions processed as well as the smart contracts recorded on the blockchain. The Ethereum blockchain will suffer damage if an Ethereum smart contract is vulnerable. In this paper, several contract vulnerabilities are detected. Thirteen contract vulnerabilities, and their corresponding descriptions are shown in Table 1.

Machine Learning

Machine learning is a method that empowers a machine to learn the underlying characteristics of data and perform functions that cannot be done by direct programming. In a practical sense, machine learning is a method of training a model from data, making the model possess the underlying patterns in the data, and finally predicting the untrained data from the trained model. Machine learning is divided into two phases: training and prediction. The training phase takes the data as input to learn the features between the data and optimizes the internal parameters of the model using the objective function as a benchmark to achieve the desired loss target. The prediction phase takes data that have not been involved in training as input and predicts the input using the model and its parameters learned in the training phase. There are two types of machine learning. One type is supervised learning, which involves the training of existing training samples to obtain an optimal model and using this model to map all the inputs to the corresponding outputs to complete the task of prediction and classification. The data in supervised learning are labeled in advance, and its training samples contain both features and label information. Common supervised learning algorithms are linear regression algorithms, BP neural network algorithms, decision trees, regression trees, logistic regression, support vector machines, KNN, and others. The other type of learning is unsupervised learning, where the training samples are unlabeled, and the goal is to reveal commonalities between data and intrinsic patterns by learning from unlabeled training samples. The most popular machine learning method in recent years is deep learning, which is based on neural networks and uses a large number of operations to extract features between data in a black-box fashion to continuously approximate an optimized objective function. The aim in this paper is to develop a machine learning model that learns vulnerabilities in

Table 1.
Thirteen Types of Security Vulnerabilities in Smart Contracts

Types of Vulnerabilities	Descriptions
Underflow	Integer underflow
Overflow	Integer overflow
CallDepth	Use send or call cmd, but do not check the cmd result
TOD	State will depend on the txorder
TimeDep	State will depend on the timestamp
Reentrancy	Contract contains reentrancy function
AssertFail	Contract contains the condition of assert fail
TxOrigin	Contract use tx.origin
CheckEffects	Contract checks if the state has been updated before the transaction or not
InlineAssembly	Contract uses assembly code
BlockTimestamp	Contract uses block timestamp
LowLevelCalls	Contract uses send or call not transfer
SelfDestruct	Contract uses self destruct

smart contracts so that it can be used to detect vulnerabilities in untrained smart contracts, making its detection universal and robust.

Natural Language Processing

Natural language processing refers to utilizing computers to process human language so that computers can read and understand human language. Natural language processing is used in machine translation, speech recognition, spam filtering, information extraction, text sentiment analysis, automatic question and answer, and personalized recommendation. Words are the smallest units of language processing, and sentences are processed by stitching together the smallest units. From a syntactic or semantic point of view, phrases can also be used as the smallest unit of language processing. For a specific sentence, both words and phrases may be the smallest units for comprehension. Sometimes language processing is also performed in phrases because words in phrases lose their meaning once they are separated. For natural language processing, a simple strong splitting of sentences does not help natural language processing because words and phrases in different contexts have their own specific meanings. The difference between natural language and code is that code has syntax and structure, which need to be considered when processing code using natural language processing methods.

DESIGN OF CON2VEC

Con2Vec consists of the ByteCodeToVec model and the BugDetection module. In this section, the construction of the ByteCodeToVec model and the BugDetection module is described, and the architecture is given in Figure 2. The entire flow of the vulnerability detection methodology is presented in Figure 3.

Design Concept

The ByteCodeToVec module is composed of two parts: the OpcodeExtractor module and the Opcode2Vec module. OpcodeExtractor converts the Ethereum bytecode into an opcode sequence and uses it as input to the Opcode2Vec module. The rules of conversion are based on the bytecode to

Figure 2.
Overview of Con2Vec

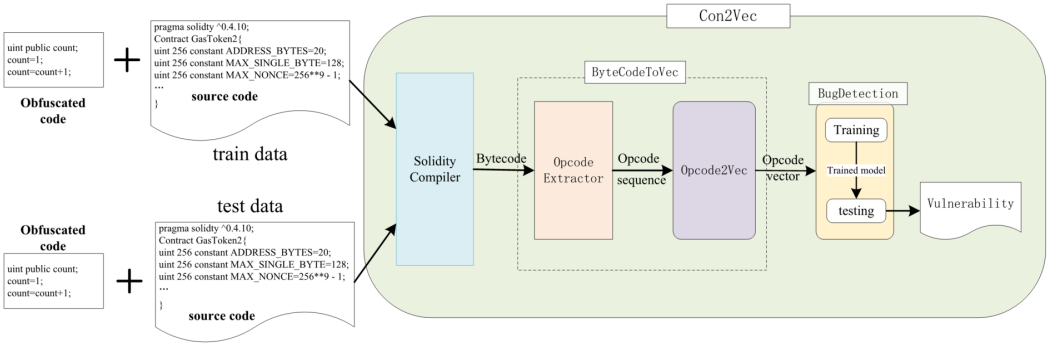
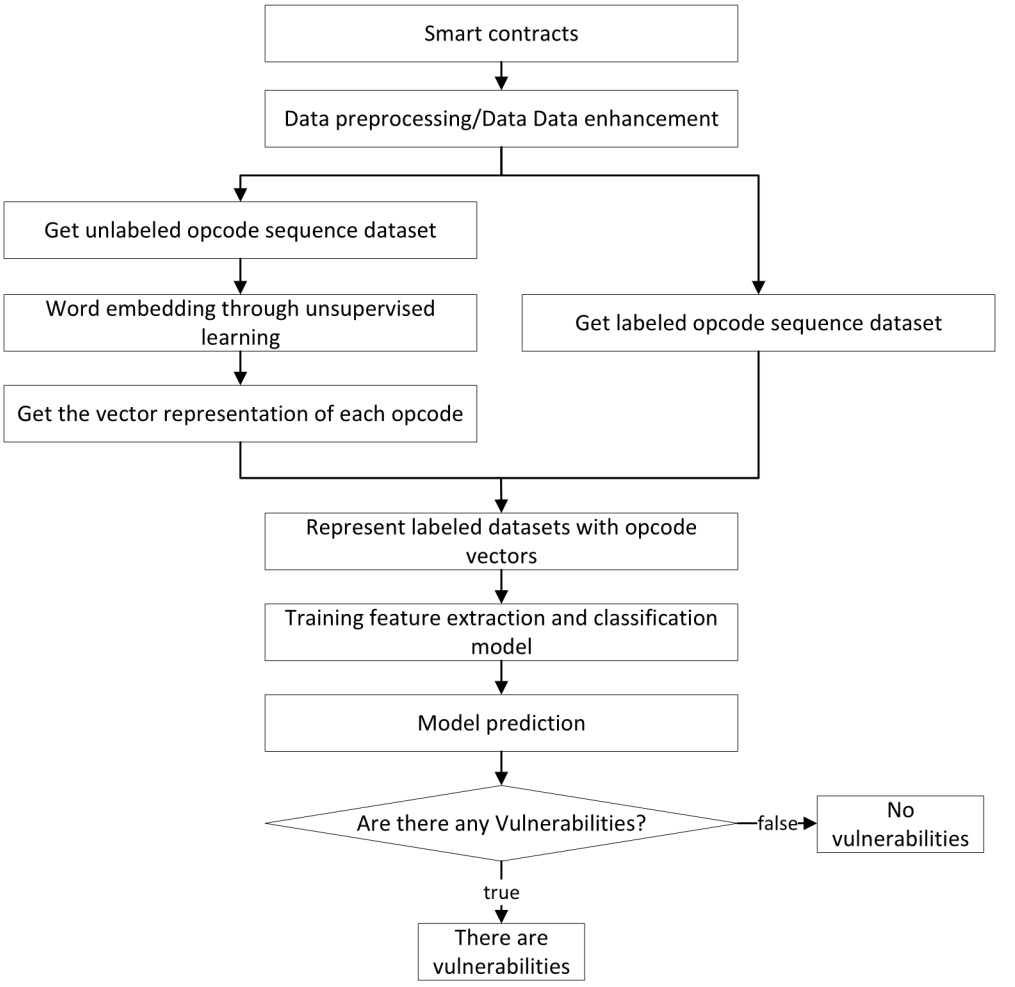


Figure 3.
The Flowchart of Vulnerability Detection



opcode correspondence rule listed in the Ethereum Yellow Book. The Opcode2Vec module mainly consists of the extended SkipGram model (Mikolov et al., 2013), which maps opcodes to vectors. The Opcode2Vec module does not vectorize natural language, as is the case with word embedding in natural language processing, but vectorizes code, so ByteCodeToVec can implicitly extract smart contract features. Con2Vec uses the code embedding information from the trained neural network as the basis for the vulnerability detection model to learn the code embedding features to detect vulnerabilities. When a smart contract vulnerability needs to be detected, the model simply learns the features of the smart contract containing the vulnerability and then feeds the opcode sequence of the smart contract to be detected into the trained detection model to output the corresponding vulnerability. To clarify, Con2Vec can detect vulnerabilities in smart contracts even if the vulnerability characteristics of the specific smart contract are not known. Con2Vec takes the EVM bytecode or contract source code to be detected as input and returns a list of detected smart contract vulnerabilities.

The BugDetection module mainly consists of TextCNN (Chen, 2015). When the ByteCodeToVec module is trained, it generates a vector representation of Ethereum smart contract opcodes. In the BugDetection module, each data input to the model is an opcode sequence. Each opcode sequence is converted from the smart contract source code and represented as a matrix of smart contract opcodes stitched together in rows of opcode vectors.

Building Blocks of ByteCodeToVec

The ByteCodeToVec model consists of the OpcodeExtractor module and the Opcode2Vec module.

OpcodeExtractor Module

The OpcodeExtractor module is a module that provides input to the Opcode2Vec module. Specifically, the ByteToOpcode module parses Ethereum bytecodes to generate opcodes according to established rules (Table 2 lists some of the rules for the mapping table of bytecodes to opcodes). According to the corresponding generation rules, the OpcodeExtractor module maps the input bytecode string in groups of two characters to obtain the corresponding opcode. It is important to note that the PUSH instruction set has its own set of rules. It consists of 32 instructions from PUSH1 to PUSH32, with each PUSH instruction followed by a number representing an additional character to be read. When the module reads a PUSH instruction, it automatically recognizes the number following the PUSH instruction, reads an additional group of characters of that size according to that value, and then uses it as the parameter value for the PUSH instruction. For example, when the module reads a certain group of input characters whose corresponding operation code is PUSH4 in accordance with the relational mapping table, the module will read 4 additional groups of characters (i.e., 8 characters, 2 characters for 1 group) and place these 4 groups of characters after the PUSH4 instruction at the same time. For other instructions, the module will convert them to opcodes as normal according to the relational mapping table. The result of converting the EVM bytecodes 0x6d4946c0e9f43f4dee607b0ef1fa1c3318585733ff to opcode is shown in Table 3.

This paper proposes three different methods of generating opcode sequences by studying the above conversion rules for bytecodes and opcodes. The first method is to remove all the operands and keep only the opcode and to concatenate all the opcodes to form an opcode sequence. The second method is to retain both opcodes and operands and to unify the operands after the opcode as NUM. For example, PUSH1 49 becomes PUSH1 NUM, PUSH2 49 46 becomes PUSH2 NUM, and so on. The third method also retains both the opcode and the operand, but it differs from the second method in that the opcode followed by the operand is unified into a single field. For example, PUSH1 49 is unified as PUSH1 NUM, PUSH2 49 46 is unified as PUSH2 NUM, and so on. Three opcode sequence generation methods are shown in Table 4.

Opcode2Vec Module

The Opcode2Vec module can vectorize smart contract code. It is a SkipGram model that obtains an opcode vector by training a sequence of opcodes. The training process is shown in Figure 4.

Table 2.
Relationship Mapping Table Between Bytecode and Opcode

Bytecode	Opcode
0x00	STOP
0x01	ADD
0x02	MUL
0x03	SUB
0x04	DIV
0x05	SDIV
0x06	MOD
0x07	SMOD
0x08	ADDMOD
0x09	MULMOD

Table 3.
Conversion Results of Smart Contract Bytecodes and Opcode

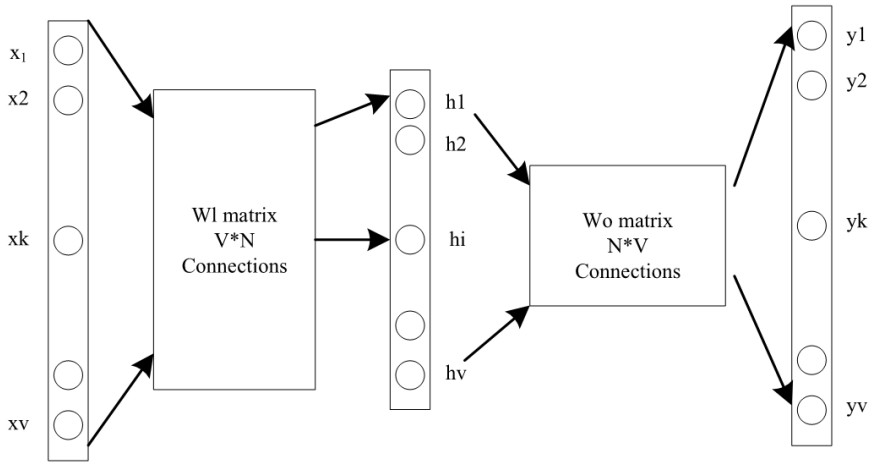
Bytecodes	Opcode And Operand
6d 49 46 c0 e9 f4 3f 4d ee 60 7b 0e f1 fa 1c	PUSH14 49 46 c0 e9 f4 3f 4d ee 60 7b 0e f1 fa 1c
33	CALLER
18	XOR
58	PC
57	JUMPI
33	CALLER
ff	SELFDESTRUCT

Table 4.
Three Methods of Generating Opcode Sequence

Method of Generating Opcode Sequences	Bytecodes	Opcode Sequence
Method 1 (Based on Opcode)	0x6d4946c0e9f43f4dee607b0ef1fa1c3318585733ff	PUSH14 CALLER XOR PC JUMPI CALLER SELFDESTRUCT
Method 2 (Based on OpcodeAndNum)	0x6d4946c0e9f43f4dee607b0ef1fa1c3318585733ff	PUSH14 NUM CALLER XOR PC JUMPI CALLER SELFDESTRUCT
Method 3 (Based on OpcodeNum)	0x6d4946c0e9f43f4dee607b0ef1fa1c3318585733ff	PUSH14NUM CALLER XOR PC JUMPI CALLER SELFDESTRUCT

The model allows a vectorized representation of the code. Specifically, if given a string of smart contract bytecodes, the OpcodeExtractor module processes the corresponding sequence of opcodes, the SkipGram model slides over the sequence of opcodes with a fixed window size, and the central opcode predicts the surrounding opcodes to generate a number of tasks. The sliding window starts at the beginning of the opcode sequence and moves forward by one opcode size at each step, which

Figure 4.
SkipGram Model



is similar to performing a multiclass inference task where each input yields a different output, and the weight matrix passed between each input to the output is different. This enables the central opcode matrix from the training process to be used as the final code vector matrix. For sliding windows, the range covered by each slide is the range in the opcode sequence, (i.e., all the predicted opcodes are in the opcode sequence). For example, if the sliding window size is 3 (3 before and 3 after the center opcode) and the current center opcode is the first opcode in the opcode sequence, then the window will only cover the three opcodes to the right of the center opcode, but not the three to the left. Thus, the left three are beyond the opcode sequence. In terms of model details, the model is trained by a neural network, and the target words are inferred from the vocabulary by the SoftMax function. The SkipGram architecture consists of a hidden layer. For example, given a vector of inputs (typically a one-hot vector), the model is trained to obtain the corresponding output, and the output layer is computed based on the SoftMax function to produce the classification result. Assuming a set of opcode sequences $\left[\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3, ..., \boldsymbol{w}_T\right]$, the objective function of the model is:

$$L = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c}\log p(w_{t+j} \mid w_t)$$

The parameter $\boldsymbol{c}$ is the size of the contextual window. The larger the value of $\boldsymbol{c}$, the more training samples are obtained, the higher the result accuracy is, but the longer training takes. The SkipGram model is defined using the SoftMax function $\boldsymbol{p}(\boldsymbol{w}_o \mid \boldsymbol{w}_i)$:

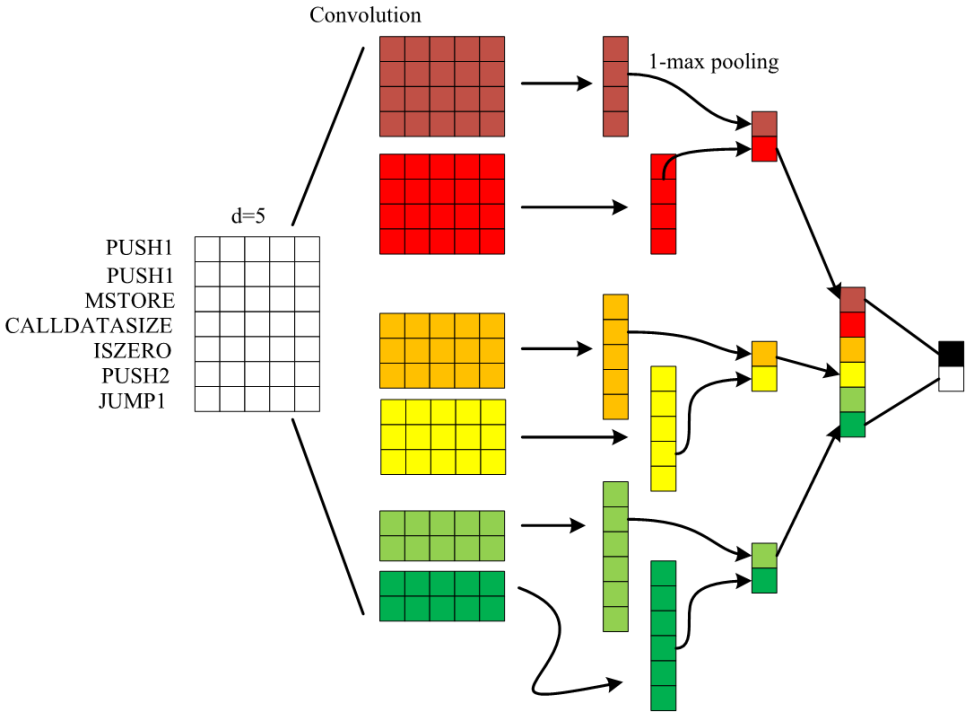
$$p(w_o \mid w_i) = \frac{\exp(v_{w_o}^{'\ T} v_{w_I})}{\sum_{w=1}^{W}\exp(v_{w_o}^{'\ T} v_{w_I})}$$

The parameters $\boldsymbol{v}_{\boldsymbol{w}}^{'}, \boldsymbol{v}_{\boldsymbol{w}}$ denote the output vector and input vector of opcode $\boldsymbol{w}$, respectively.

Building Blocks of BugDetection

For the ByteCodeToVec model, the training objective is to optimize the opcode vector belonging to each smart contract, where opcodes with similar meanings are mapped to similar locations in the vector space. The opcode vectors obtained by this model can be invoked as features of the vector representation for contract-level analysis. The algorithm is unsupervised, so the process does not need to label each contract in advance. The features obtained from this training can be utilized in BugDetection, a module that detects smart contract vulnerabilities through model training. Specifically, the main body of the module consists of a TextCNN, with the input of the model being the opcode matrix and the output being the presence or absence of vulnerabilities. The scheme generally uses the CNN-multichannel method. The embedding layer is initialized by the pretrained SkipGram vector, and the opcodes that do not appear in the pretrained word vector, due to their version, are randomly initialized. From there, the embedding layer is fixed, and the whole network is fine-tuned to obtain the fine-tuned word vector. Then, together with the pretrained SkipGram vector, multiple channels are formed simultaneously to learn the features of both embeddings during the model training process. In the convolutional pooling layer, convolution is performed only for the vertical direction of the opcode sequence, (i.e., the width of the convolutional kernel is fixed to the dimension of the opcode vector). Convolutional kernels of different heights can be used for convolution, resulting in richer feature representations and indirectly more N-gram features. The pooling layer can choose 1-max pooling to extract the maximum value in the feature map. By choosing the maximum value of each feature map, the most important features in each feature map can be captured. Each convolutional kernel will obtain an important feature value. 1-max pooling is used for all convolutional kernels and then cascaded to obtain the final feature vector, which is fed into the SoftMax layer for binary classification to obtain the vulnerability detection model. The TextCNN model for vulnerability detection is illustrated in Figure 5.

Figure 5.
Vulnerability Detection Model



Multilabel Classification Problems

For a labeled smart contract dataset, a smart contract may have multiple vulnerabilities, meaning that each contract may have multiple labels, each label corresponding to one vulnerability. The detection tool in this paper provides detection of the following 13 smart contract vulnerabilities: underflow vulnerability (marked as C1), overflow vulnerability (marked as C2), CallDepth vulnerability (marked as C3), TOD vulnerability (marked as C4), TimeDep vulnerability (marked as C5), reentrancy vulnerability (marked as C6), AssertFail vulnerability (marked as C7), TxOrigin vulnerability (marked as C8), CheckEffects vulnerability (marked as C9), InlineAssembly vulnerability (marked as C10), BlockTimeStamp vulnerability (marked as C11), LowLevelCalls vulnerability (marked as C12), and SelfDestruct vulnerabilities (marked as C13). In theory, this research method is applicable to the detection of various smart contract vulnerabilities, provided that a considerable number of samples of the vulnerable smart contract dataset are available for training. Examples of vulnerability labels are shown in Table 5. The label [1,0,0,0,0,0,0,0,0,0,0,0,1] for the smart contract sample X1 indicates that the contract has both underflow vulnerability and SelfDestruct vulnerability, free of the other 11 additional vulnerabilities. To address the multilabel classification problem in vulnerability mining, this paper uses a one-to-rest (one vs rest) strategy to classify the dataset into two categories for each label, namely, the presence of the vulnerability and the absence of the vulnerability, thus converting the multilabel classification problem into a multiple binary classification problem.

Data Imbalance Problems

The training effect can be afflicted when there is a disparity between the proportion of positive and negative classes of vulnerabilities for smart contract datasets. In a naturally distributed Ethereum smart contract, the proportion of vulnerabilities that occur is expected to be low. Assuming a total of 100 training data points, and 99 of them do not contain reentrancy vulnerabilities, it is clear that if these data are trained, the trained model has a 99% accuracy rate of predicting whether a particular smart contract is in the negative class (i.e., there are almost no reentrancy vulnerabilities), but such a model often has no value. Therefore, to address the data imbalance problem in the smart contract dataset, this paper proposes a code obfuscation method for smart contracts to address the smart contract dataset imbalance problem, and to improve the redundant code interference resistance of this detection model. The idea of the method is to prepare a number of redundant codes for obfuscation in advance and then randomly insert the redundant codes into the smart contracts according to the rules before compiling the smart contracts in the dataset to form a new smart contract with redundant codes. Specifically, the redundant code can include either simple computation of invalid variables (i.e., variables that cannot be defined in the same way as the original smart contract code and are limited to defining some useless variables for self-operation) or invalid functions that do not affect the functionality of the smart contract system. The specific code obfuscation method is shown in Algorithm 1. Line 1 indicates that the location of all right brackets in the smart contract source code is found and marked, returning an array of location markers, which stores the location of the right brackets in the smart contract source code. Line 2 indicates the number of right brackets for all functions in the source code of the smart contract. Line 3 indicates that the number of redundant

Table 5.
Examples of a Vulnerability Label

Contract	Contract Bytecodes	Label
Contract X1	0x6d4946c0e9f43f4dee607b0ef1fa1c3318585733ff	[1,0,0,0,0,0,0,0,0,0,0,0,1]
Contract X2	0x6080604052348015610010576000080fd	[1,0,0,0,0,1,0,0,0,0,0,0,0]
Contract X3	0x6060604052361561006c5760e060020a60003504	[1,0,1,0,0,1,1,1,1,1,1,0,1]

codes to be embedded is calculated. Line 4 defines an array that receives the redundant codes (i.e., where the redundant code in the source code is inserted), with an array size of countRandomPosition. Line 5, line 6, line 7, and line 8 indicate the random selection of countRandomPosition right bracket positions to insert the redundant code. Line 9 indicates the random shuffle of the redundant code to be inserted. Line 10 indicates that the prepared redundant code is inserted one by one in an array order into the next line of the randomly selected source code corresponding to the right bracket position. Line 11 returns the smart contract with the redundant code embedded. With this algorithm, a large number of nonredundant smart contracts can be generated, thus addressing the issue of data imbalance. Moreover, the detection model is also resistant to regular code redundancy, as a large amount of redundant code is randomly inserted into the normal code.

```
Algorithm 1:
Algorithm 1: RandomInsertion
Input: contractFile, redundantcode[]
Output: contractFile
 1: rightBracketPosition[]=FindFuncRightBracket(contractFile)
 2: count=len(rightBracketPosition)
 3: countRandomPosition=len(reddundantCode)
 4: Define an integer array named writeRandomPostion to store
randomly generated insertion position
 5: while countRandomPosition>0 do
 6:       writeRandomPosition.append(random.randInt(0,count))
 7:       countRandomPosition--
 8: end while
 9: shuffle(redudantcode)
10: wRddtCodeToContractFile(writeRandomPosition,redundantCode)
11: return contractFile
```

EXPERIMENT

Purpose of the Experiment

To evaluate the performance of Con2Vec, vulnerabilities in the smart contract code to be analyzed can be detected by training a smart contract that is known to contain vulnerabilities. To do this, it is first necessary to check whether ByteCodeToVec appropriately represents the relationship between the smart contract code to be analyzed and the smart contract code learned during the training phase. Based on this, a number of smart contracts written by Solidity containing integer overflow vulnerabilities were predesigned and evaluated against these known vulnerable smart contracts to confirm whether ByteCodeToVec could properly extract the features of the code. Next, the smart contract code with known vulnerabilities was trained and evaluated with TextCNN to check whether the model successfully extracted the features of the smart contract and identified all the vulnerabilities learned. Through these experiments, it can be seen that ByteCodeToVec is able to accurately extract features and thus detect vulnerabilities. In the field of vulnerability detection of Ethereum smart contract, datasets are not provided in most papers, while data sets are provided in the SoliAudit (Liao et al., 2019). SoliAudit's method is effective and representative among the current static vulnerability detection methods. Therefore, this paper conducts experiments on the dataset of SoliAudit and compares the performance of Con2Vec with that of SoliAudit, and the experiments prove that Con2Vec outperforms SoliAudit.

Experimental Setup

In the mapping phase for smart contract opcodes, the SkipGram model is used for training, and the samples are negatively sampled during the training process to speed up the training. In the

vulnerability detection module, TextCNN is used in the main body for feature extraction and vulnerability classification, and pretrained opcode vectors are added to fine-tune the model during the training phase. The convolution kernel sizes are set to 3, 4, and 5, corresponding to the trigram, 4-gram, and 5-gram in the language model, respectively, and 1-max pooling is used to obtain the most key features in the opcode sequence. The experimental results are measured using accuracy, precision, recall, and F1-score.

Datasets

This experiment obtains a total of over 60,000 unlabeled smart contracts from the Ethereum platform to build a dataset, which are all open source on the Ethereum platform and whose contracts are real and valid. In addition, this paper cleans and merges the public datasets from authoritative papers to obtain over 17,000 labeled smart contract datasets, most of which are imbalanced. The number of positive and negative samples for each of the original dataset vulnerabilities (positive samples represent vulnerabilities, negative samples represent no vulnerabilities) is shown in Table 6. A code obfuscation method is given to expand the 17,000 labeled datasets to 40,000, where the positive to negative ratio is 1:1. The datasets are publicly available at https://github.com/starStraw/OpcodeToVec.

RESULTS

The vulnerability detection results of Con2Vec are shown in Tables 7, 8, and 9. Table 7 presents the results of vulnerability detection based on Opcode sequence (OpSe), Table 8 shows the results of vulnerability detection based on OpcodeAndNum sequence (OpANSe), and Table 9 shows the results of vulnerability detection based on OpcodeNum sequence (OpNSe).

As shown in Table 7, the prediction accuracy of the model for underflow, overflow, TimeDep, AssertFail, InlineAssembly, BlockTimeStamp, and LowLevelCalls vulnerabilities all reached more than 97%, which is higher than the prediction accuracy of other vulnerabilities.

As shown in Table 8, the prediction accuracy of the model for underflow and overflow vulnerabilities reached more than 97%, which is higher than the prediction accuracy of other vulnerabilities.

Table 6.
Number of Positive and Negative Samples for Each Vulnerability in the Dataset

Types of Vulnerability	Positive (With Vulnerability)	Negative (Without Vulnerability)
Underflow	10883	6527
Overflow	15499	1911
CallDepth	386	17024
TOD	2523	14887
TimeDep	1175	16235
Reentrancy	513	16897
AssertFail	7450	9960
TxOrigin	144	17266
CheckEffects	6853	10557
InlineAssembly	1286	16124
BlockTimeStamp	5625	11785
LowLevelCalls	5153	12257
SelfDestruct	1255	16155

Table 7.
Vulnerability Detection Results of OpSe

Types of Vulnerability	Accuracy	Precision	Recall	F1-score
Underflow	97.08%	97.37%	96.77%	97.07%
Overflow	97.24%	97.93%	96.52%	97.22%
CallDepth	96.34%	96.54%	96.13%	96.33%
TOD	96.37%	96.77%	95.94%	96.35%
TimeDep	97.11%	98.05%	96.13%	97.08%
Reentrancy	95.83%	95.40%	96.30%	95.85%
AssertFail	97.52%	97.66%	97.37%	97.52%
TxOrigin	95.92%	96.50%	95.30%	95.89%
CheckEffects	95.59%	96.15%	94.98%	95.56%
InlineAssembly	97.65%	98.84%	96.43%	97.62%
BlockTimeStamp	97.48%	98.96%	95.97%	97.44%
LowLevelCalls	97.30%	98.13%	96.44%	97.28%
SelfDestruct	96.48%	97.81%	95.09%	96.43%
Average	96.76%	97.39%	96.11%	96.74%

Table 8.
Vulnerability Detection Results of OpANSe

Types of Vulnerability	Accuracy	Precision	Recall	F1-score
Underflow	97.25%	97.55%	96.93%	97.24%
Overflow	97.12%	97.63%	96.58%	97.10%
CallDepth	95.83%	96.00%	96.13%	96.33%
TOD	95.87%	96.24%	95.47%	95.85%
TimeDep	96.46%	96.97%	95.92%	96.44%
Reentrancy	96.21%	96.39%	96.02%	96.20%
AssertFail	96.94%	98.47%	95.36%	96.89%
TxOrigin	95.79%	95.91%	95.66%	95.78%
CheckEffects	96.08%	96.93%	95.17%	96.04%
InlineAssembly	96.58%	96.95%	96.19%	96.57%
BlockTimeStamp	96.52%	97.18%	95.82%	96.50%
LowLevelCalls	96.78%	97.48%	96.04%	96.76%
SelfDestruct	96.25%	97.09%	95.36%	96.22%
Average	96.44%	96.98%	95.90%	96.46%

As shown in Table 9, the prediction accuracy of the model for overflow, TimeDep, AssertFail, InlineAssembly, and BlockTimeStamp vulnerabilities reached more than 97%, which is higher than the prediction accuracy of other vulnerabilities.

Table 9.
Vulnerability Detection Results of OpNSe

Types of Vulnerability	Accuracy	Precision	Recall	F1-score
Underflow	96.91%	97.36%	96.43%	96.90%
Overflow	97.05%	97.91%	96.15%	97.02%
CallDepth	96.28%	96.34%	96.22%	96.28%
TOD	96.08%	96.37%	95.77%	96.07%
TimeDep	97.11%	98.05%	96.13%	97.08%
Reentrancy	95.92%	96.43%	95.37%	95.90%
AssertFail	97.34%	98.04%	96.61%	97.32%
TxOrigin	96.23%	96.57%	95.86%	96.22%
CheckEffects	95.66%	96.05%	95.24%	95.64%
InlineAssembly	97.32%	98.09%	96.51%	97.30%
BlockTimeStamp	97.02%	97.77%	96.24%	97.00%
LowLevelCalls	96.89%	97.77%	95.97%	96.86%
SelfDestruct	96.13%	97.00%	95.21%	96.09%
Average	96.61%	97.21%	95.98%	96.59%

According to the data in Figures 6, 7, and 8, it can be found that the average accuracy of Con2Vec in detecting various vulnerabilities in the current dataset is up to 96% or more, which is higher than the detection method in SoliAudit with an accuracy of 91.4%. Even when compared to vulnerabilities that occur infrequently, the recall rate of Con2Vec is more stable and higher than that of SoliAudit's method. It can be seen that the Con2Vec method is more efficient in detection in Table 10. When the model is trained and the smart contract to be detected is fed into the model, the result is obtained in less than 0.1 s (with batch testing, it takes less than two minutes to test 1,700 smart contracts).

As shown in Figure 9, the histogram shows the comparison of accuracy, precision, recall and F1-score of the three methods. From the figure, it can be seen that the overall metrics of the OpSe method are better than those of the other two among the above three detection methods. Among them, the OpANSe method is slightly worse than the other two, while the OpSe method and the OpNSe method are not much different in all aspects, probably because of the approximation of the converted method when converting bytecode to opcode sequence (OpcodeNum sequence). In its conversion, the OpSe method discards all operands and retains only the opcode. The OpNSe method retains the operands but does not have a significant effect on the overall result because only opcodes such as PUSH are followed by operands. Although the names of the converted opcodes are different, the final result is similar, with only a few opcodes to distinguish between them.

DISCUSSION

It is found that when the dataset is expanded to a certain level, the detection effect of using the similarity between opcodes to simplify the opcodes to generate the opcode sequences is lower than the result based on opcode sequences alone, which may be because after the dataset is expanded, the code embedding method is better able to find the subtle differences between the opcodes without the need to simplify the opcodes to focus features and thus find differences. Additionally, unknown vulnerabilities are not managed by the method proposed in this paper because the dataset must be labeled before training. Despite the limitations, the method can retrain and detect the vulnerabilities

Figure 6.
Comparison Between OpSe and SoliAudit

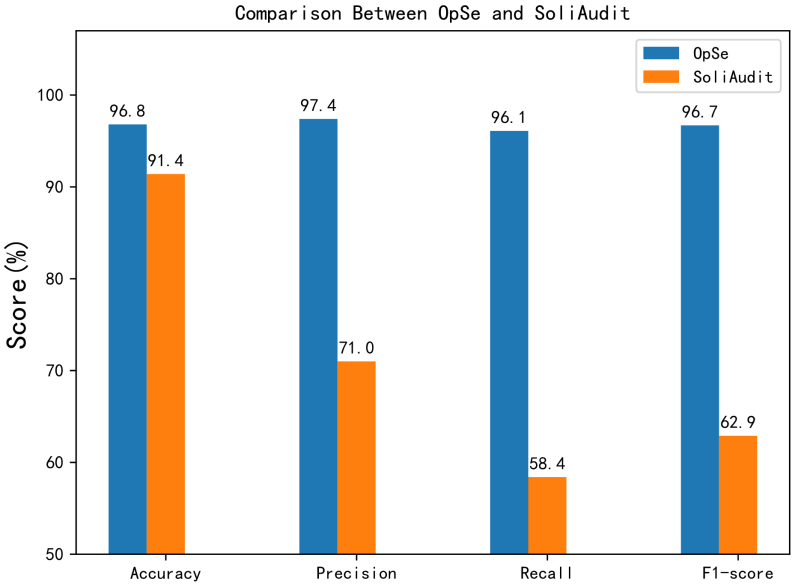
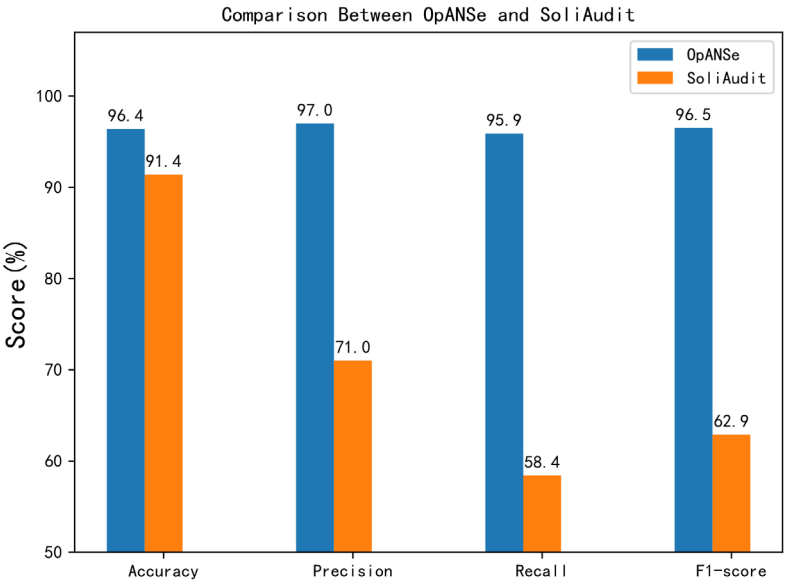


Figure 7.
Comparison Between OpANSe and SoliAudit



as long as there are enough labeled smart contract samples with new vulnerabilities. In the process of researching the area of smart contract vulnerability detection, it is found that the current smart contract datasets are less publicly available, and there is no large-scale smart contract corpus. Due to the different domains, the current large-scale pretrained models cannot be used in the area of smart

Figure 8.
Comparison Between OpNSe and SoliAudit

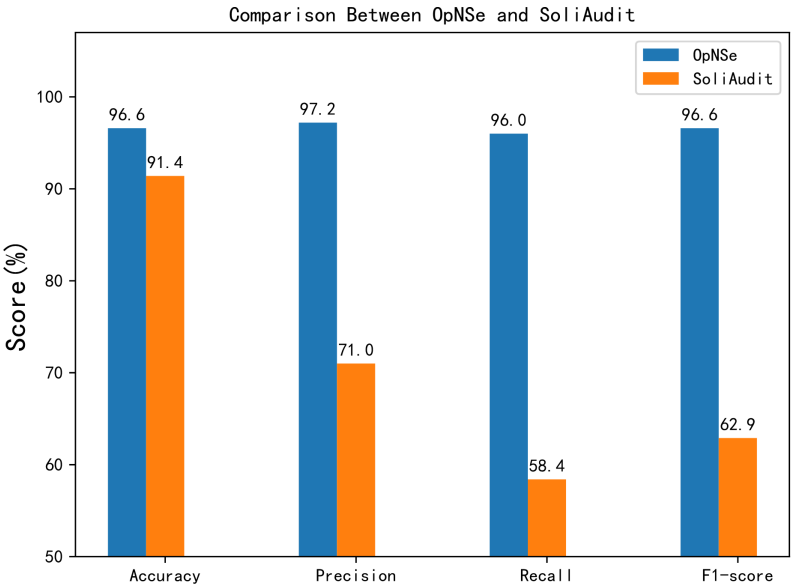
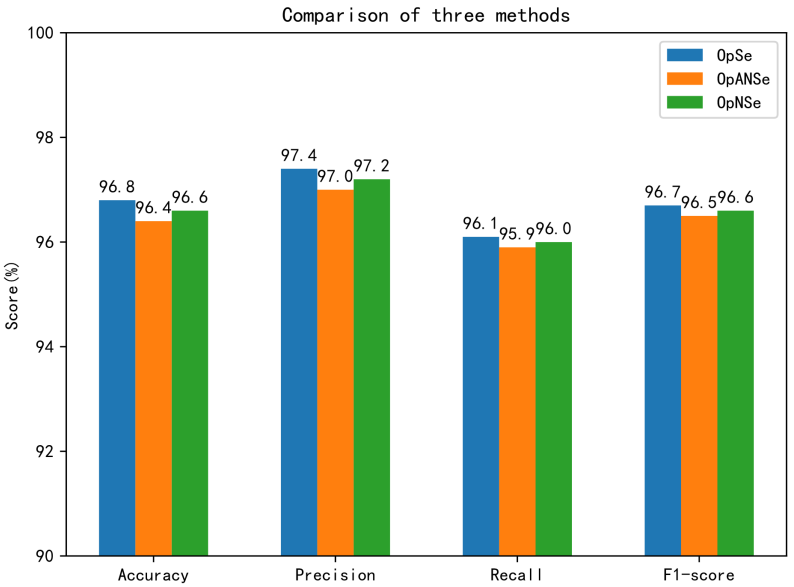


Table 10.
Comparison of Average Time Consumption of Different Methods (Seconds)

Variable	OpSe	OpANSe	OpNSe	Oyente	Mythril
Time	<=0.1	<=0.1	<=0.1	22	3612

Figure 9.
Comparison of Three Methods



contract vulnerability detection, so the current static detection for smart contract vulnerabilities mostly uses simple deep learning models and machine learning models.

CONCLUSION

This paper proposes a vulnerability detection method based on embedding and TextCNN model for smart contracts. Through collating and cleaning existing datasets, over 60,000 unlabeled smart contract datasets are disclosed, and a code obfuscation method for data enhancement of labeled smart contracts is offered. This method can effectively generate enough obfuscation samples, enhancing the capability of anti-redundant code for the detection model. Three different methods for generating opcode sequences are also proposed in this paper, among which the vulnerability detection results based on opcode sequence is optimal. A total of 13 smart contract vulnerabilities are detected in this paper with reliable performance in terms of accuracy. The highlight of the model is that it converts the smart contract vulnerability detection problem into a natural language text classification problem with a short detection time and low miss rate. The result shows that the average accuracy of this method for smart contract vulnerability detection is above 96% and no prior knowledge is required. This approach could theoretically also be applied to other types of smart contracts, such as fabric smart contracts. Fabric smart contract is essentially a program written in Golang language. As long as the Golang code is converted into opcodes, and then feature extraction and model training are performed on the opcode using this method, the vulnerability of the fabric smart contract can be easily detected. In future work, solutions in the area of smart contract vulnerability detection can be extended by improving the vector representation of smart contracts.

ACKNOWLEDGMENT

Li Zhao and Qi Li helped proofread manuscripts. We sincerely thank the anonymous reviewers for their very comprehensive and constructive comments.

CONFLICT OF INTEREST

The authors of this publication declare there is no conflict of interest.

FUNDING STATEMENT

This work is supported by the National Natural Science Foundation of China, under grant No.61772180, the Key-Area Research and Development Program of Guangdong Province 2020B1111420002, the Key Research and Development Project in Hubei Province 2022BAA040, the Science and Technology Project of Department of Transport of Hubei Province 2022-11-4-3, and the Innovation Fund of Hubei University of Technology. BSQD2019027, BSQD2019020, and BSQD2016019.

REFERENCES

Ashizawa, N., Yanai, N., Cruz, J. P., & Okamura, S. (2021). *Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts* [Paper presentation]. *3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*. doi:10.1145/3457337.3457841

Chen, R., Wang, Z., & Hong, Y. (2021). Pipelined XPath query based on cost optimization. *Scientific Programming*, *2021*, 1–16. doi:10.1155/2021/5089236

Chen, Y. (2015). *Convolutional neural network for sentence classification*. University of Waterloo.

Chinen, Y., Yanai, N., Cruz, J. P., & Okamura, S. (2020). *RA: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis* [Paper presentation]. *2020 IEEE International Conference on Blockchain (Blockchain)*. doi:10.1109/Blockchain50366.2020.00048

Ding, M., Li, P., Li, S., & Zhang, H. (2021). Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. *Evaluation and Assessment in Software Engineering,* 321-328.

Durieux, T., Ferreira, J. F., Abreu, R., & Cruz, P. (2020). *Empirical review of automated analysis tools on 47,587 ethereum smart contracts* [Paper presentation]. ACM/IEEE 42nd International Conference on Software Engineering.

Eshghie, M., Artho, C., & Gurov, D. (2021). Dynamic vulnerability detection on smart contracts using machine learning. *Evaluation and Assessment in Software Engineering*, 305-312.

Fu, X., Wang, H., & Shi, P. (2021). A survey of blockchain consensus algorithms: Mechanism, design and applications. *Science China. Information Sciences*, *64*(2), 121101. Advance online publication. doi:10.1007/s11432-019-2790-1

Grieco, G., Song, W., Cygan, A., Feist, J., & Groce, A. (2020). *Echidna: effective, usable, and fast fuzzing for smart contracts* [Paper presentation]. *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3395363.3404366

Holub, M., & Johnson, J. (2019). The impact of the bitcoin bubble of 2017 on bitcoin's P2P market. *Finance Research Letters*, *29*, 357–362. doi:10.1016/j.frl.2018.09.001

Jiang, B., Liu, Y., & Chan, W. K. (2018). *Contractfuzzer: Fuzzing smart contracts for vulnerability detection* [Paper presentation]. *33rd ACM/IEEE International Conference on Automated Software Engineering*. doi:10.1145/3238147.3238177

Kowalski, M., Lee, Z. W., & Chan, T. K. (2021). Blockchain technology and trust relationships in trade finance. *Technological Forecasting and Social Change*, *166*, 120641. doi:10.1016/j.techfore.2021.120641

Li, D., Han, D., Weng, T.-H., Zheng, Z., Li, H., Liu, H., Castiglione, A., & Li, K.-C. (2022). Blockchain for federated learning toward secure distributed machine learning systems: A systemic survey. *Soft Computing*, *26*(9), 4423–4440. doi:10.1007/s00500-021-06496-5 PMID:34840525

Liao, J.-W., Tsai, T.-T., He, C.-K., & Tien, C.-W. (2019). *Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing* [Paper presentation]. *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. doi:10.1109/IOTSMS48152.2019.8939256

Liu, Y., Zhang, J., Wu, S., & Pathan, M. S. (2021). Research on digital copyright protection based on the hyperledger fabric blockchain network technology. *PeerJ. Computer Science*, *7*, e709. doi:10.7717/peerj-cs.709 PMID:34616889

Lutz, O., Chen, H., Fereidooni, H., Sendner, C., Dmitrienko, A., Sadeghi, A. R., & Koushanfar, F. (2021). ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *arXiv preprint arXiv:2103.12607*. https://arxiv.org/abs/2103.12607

Mehar, M. I., Shier, C. L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H. M., & Laskowski, M. (2019). Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *Journal of Cases on Information Technology*, *21*(1), 19–32. doi:10.4018/JCIT.2019010102

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. https://arxiv.org/abs/1301.3781

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review,* 21260.

Park, Y.-H., Kim, Y., Lee, S.-O., & Ko, K. (2021). Secure outsourced blockchain-based medical data sharing system using proxy re-encryption. *Applied Sciences (Basel, Switzerland)*, *11*(20), 9422. doi:10.3390/app11209422

Pournader, M., Shi, Y., Seuring, S., & Koh, S. L. (2020). Blockchain applications in supply chains, transport and logistics: A systematic review of the literature. *International Journal of Production Research*, *58*(7), 2063–2081. doi:10.1080/00207543.2019.1650976

Praitheeshan, P., Pan, L., Yu, J., Liu, J., & Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*. https://arxiv.org/abs/1908.08605

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). *Smartcheck: Static analysis of ethereum smart contracts* [Paper presentation]. *1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. doi:10.1145/3194113.3194115

Torres, C. F., Iannillo, A. K., Gervais, A., & State, R. (2021). *Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts* [Paper presentation]. 2021 IEEE European Symposium on Security and Privacy (EuroS&P).

Torres, C. F., Schütte, J., & State, R. (2018). *Osiris: Hunting for integer bugs in ethereum smart contracts* [Paper presentation]. *34th Annual Computer Security Applications Conference*.

Wang, W., Song, J., Xu, G., Li, Y., Wang, H., & Su, C. (2020). Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, *8*(2), 1133–1144. doi:10.1109/TNSE.2020.2968505

Wang, Z., Wang, T., Hu, H., Gong, J., Ren, X., & Xiao, Q. (2020). Blockchain-based framework for improving supply chain traceability and information sharing in precast construction. *Automation in Construction*, *111*, 103063. doi:10.1016/j.autcon.2019.103063

Weiss, K., & Schütte, J. (2019, Sep. 23-27). *Annotary: A concolic execution system for developing secure smart contracts* [Paper presentation]. Computer Security–ESORICS 2019: *24th European Symposium on Research in Computer Security*, Luxembourg.

Wüstholz, V., & Christakis, M. (2020). *Harvey: A greybox fuzzer for smart contracts* [Paper presentation]. *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3368089.3417064

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2020, July). Smart Contract Vulnerability Detection using Graph Neural Network. IJCAI, 3283-3290. doi:10.24963/ijcai.2020/454

Zhigang Xu was born in 1977. He holds a PhD and is a professor (Chutian Scholar) and Master Tutor. His main research interests include blockchain, distributed system security and big data governance.

Xingxing Chen was born in 1998. He is a postgraduate and his main research interests include vulnerability detection of blockchain smart contract and knowledge graph.

Xinhua Dong was born in 1976. He earned a PhD and is a lecturer (Master Tutor). He is a member of the China Computer Federation. His main research interests include big data management, cloud computing, information retrieval and distributed system security.

Hongmu Han was born in 1980. He holds a PhD and is a lecturer (Master Tutor). He is a member of the China Computer Federation. His main research interests include blockchain, information security and mobile security.

Zhongzhen Yan was born in 1980. She earned a PhD and is a lecturer (Master Tutor). Her main research interests include information integration, data mining, artificial intelligence and image recognition.

Kangze Ye was born in 1998. He is a postgraduate and his main research interests include blockchain network propagation and data governance.

Chaojun Li was born in 1997. He is a postgraduate and his main research interests include vulnerability detection of blockchain smart contract.

Zhiqiang Zheng was born in 1972. He is a Master and his main research interest is information security.

Haitao Wang was born in 1981. He is a Master and his main research interest is information security.

Jiaxi Zhang was born in 1982. He is a bachelor and his main research interest is information security.