electronics

MDPI

Article

Multi-Label Vulnerability Detection of Smart Contracts Based on Bi-LSTM and Attention Mechanism

Shenyi Qian, Haohan Ning, Yaqiong He and Mengqi Chen *

School of Computer and Communication Engineering, Zhengzhou University of Light Industry, Zhengzhou 450001, China
* Correspondence: monkiq@163.com

Abstract: Smart contracts are decentralized applications running on blockchain platforms and have been widely used in a variety of scenarios in recent years. However, frequent smart contract security incidents have focused more and more attention on their security and reliability, and smart contract vulnerability detection has become an urgent problem in blockchain security. Most of the existing methods rely on fixed rules defined by experts, which have the disadvantages of single detection type, poor scalability, and high false alarm rate. To solve the above problems, this paper proposes a method that combines Bi-LSTM and an attention mechanism for multiple vulnerability detection of smart contract opcodes. First, we preprocessed the data to convert the opcodes into a feature matrix suitable as the input of the neural network and then used the Bi-LSTM model based on the attention mechanism to classify smart contracts with multiple labels. The experimental results show that the model can detect multiple vulnerabilities at the same time, and all evaluation indicators exceeded 85%, which proves the effectiveness of the method proposed in this paper for multiple vulnerability detection tasks in smart contracts.

Keywords: blockchain security; smart contract; vulnerability detection; multi-label classification

Citation: Qian, S.; Ning, H.; He, Y.; Chen, M. Multi-Label Vulnerability Detection of Smart Contracts Based on Bi-LSTM and Attention Mechanism. *Electronics* **2022**, *11*, 3260. https://doi.org/10.3390/electronics11193260

Academic Editor: Martin Reisslein

Received: 21 September 2022
Accepted: 7 October 2022
Published: 10 October 2022

1. Introduction

With the widespread application of blockchain technology [1] in different scenarios, it has moved from the blockchain 1.0 era, in which the programmable digital cryptocurrency system is the main feature, to the 2.0 era, in which the programmable financial system is the main feature [2]. Blockchain originates from the underlying technology of Bitcoin, which is essentially a distributed ledger of shared transactions using cryptography to form a special chained data structure that is jointly maintained by all miners in the blockchain network (i.e., nodes or users in the blockchain network) according to a consensus protocol [3], and its main features include decentralization, time-series data, collective maintenance, programmability, and secure and trustworthy, etc. The concept of a smart contract was first proposed by computer scientist Nick Szabo in the 20th century: "A smart contract is a set of commitments defined in digital form, including an agreement that the participants in the contract can execute these commitments" [4]. However, due to the lack of a credible execution environment, the exploration was only at the theoretical level until the emergence of blockchain technology provided a reliable execution environment for smart contracts. By integrating various technologies such as distributed data storage, smart contracts, consensus mechanisms [5], and encryption algorithms, blockchain can realize remote point-to-point value transfer without any trusted third party. A smart contract is a program deployed on the blockchain. The node makes the Ethereum virtual machine execute code by calling the interface of the smart contract to realize the execution of the smart contract.

As a representative technology of the blockchain 2.0 era, smart contracts offer broader and more practical applications than earlier blockchains designed only for cryptocurrencies. According to relevant statistics, the daily transaction volume on the Ethereum platform

at this stage can reach USD 4 billion, and smart contracts account for about 53.5% of the transactions, the high value held by smart contracts in different domains makes them a prime target for attackers. In 2016, the DAO smart contract running on the Ethereum public chain was attacked and the loss involved a total of more than USD 60 million [6]. In 2017, the multi-signature wallet smart contract that came with the Parity client was found to be vulnerable, with attackers stealing more than USD 30 million from three highly secure multi-signature contracts. In 2020, the lending protocol Lendf.Me was hacked and USD 25 million worth of assets within the contract were looted. The frequent incidents of smart contract security vulnerabilities have attracted a large number of scholars to conduct research in this area. Traditional smart contract vulnerability detection methods usually rely on hard rules or patterns defined by experts to detect smart contract vulnerability [7], and thus these methods have disadvantages such as limited application, high false alarm rate, poor scalability, and time consumption. In recent years, some researchers have proposed using machine learning and deep learning methods for smart contract vulnerability detection to automate vulnerability detection, but these methods can only use binary detection for smart contracts, and face problems such as fewer detectable types and poor interpretability for smart contracts that contain multiple vulnerabilities.

We propose a smart contract vulnerability detection method that combines Bi-LSTM and an attention mechanism for Ethereum smart contracts. The method first performs data preprocessing, parses the smart contract bytecode binary file into opcodes, and removes some opcodes to group opcodes with the same function into one category, uses the Word2Vec model to convert the opcode dataset into a feature matrix that can be used as an input to the neural network, and also uses the static analysis tool Oyente to add multi-label vectors to the contract opcodes. Then, it goes through the vulnerability detection module and implements the multi-label classification task, which consists of a Bi-LSTM layer, an attention layer, and a fully connected layer, where the Bi-LSTM layer is used to obtain the semantic relationships between the preceding and following opcodes in the contract. The attention layer adds weights to the important features to ensure the effectiveness of the model for smart contract vulnerability detection, and finally multi-label classification is realized through the fully connected layer. Our main contributions are as follows:

- We propose a model for multi-label classification of smart contract opcodes. Existing models perform dichotomous detection of contracts, which can only achieve the identification of one type of vulnerability, while our model can identify multiple vulnerabilities of smart contracts simultaneously.
- Our approach performed well on the test set, exceeding 85% for each of the four evaluation indicators, as well as exceeding 85% detection accuracy for each of the five vulnerabilities.
- We used the Word2Vec model to preprocess the opcode data and achieve higher classification accuracy compared to the NN Embedding model.

The rest of this paper is organized as follows. Section 2 details the background of Ethereum smart contracts, multi-label classification, and existing smart contract vulnerability detection methods. Section 3 describes five smart contract vulnerabilities and analyzes the relationship between contract source code, bytecode, and opcode. Section 4 explains our proposed method, and Section 5 presents experimental details and evaluation results. Section 6 concludes the paper and provides an outlook.

2. Background

2.1. Ethereum Smart Contract

A smart contract is a digital agreement that uses algorithms and programs to formulate contract terms, is deployed on the blockchain, and can be automatically executed according to the rules. Its original intention is to build smart contracts into physical entities to create various flexible and controllable intelligent assets, but due to the limitations of early computing conditions and the lack of application scenarios, smart contracts have not received extensive attention from researchers [8]. In December 2013, Buterin proposed the

Ethereum blockchain platform. In addition to realizing digital currency transactions based on the built-in Ether, it also provided a Turing-complete programming language (Solidity) to write smart contracts, thus applying smart contracts to the blockchain for the first time. Blockchain realizes decentralized storage, and smart contracts realize decentralized computing based on it [9].

With the development of smart contracts, more and more smart contracts are deployed on different blockchain platforms, such as Ethereum, Fabric, ESO, etc. [10]. Among the many blockchain systems that support the operation of smart contracts, Ethereum, as the earliest blockchain platform recognized to support the operation of smart contracts, has now become the largest blockchain platform in the world. In recent years, among the various research works on smart contracts, most of them focus on the relatively mature smart contracts of the Ethereum platform, and the research in this paper is also based on the smart contracts of Ethereum platform. Ethereum customizes Turing-complete programming languages such as Solidity and Serpent to develop smart contracts to achieve special contract functions. In Ethereum, the smart contract source code is compiled into machine-recognizable bytecode by a compiler to form a contract deployment transaction, and each successful deployment generates a new contract account, which is then packaged into blocks for multiple distribution and replication among Ethereum nodes, and finally run on the Ethereum Virtual Machine (EVM), which is the core of smart contract operation and is mainly used to handle the deployment and execution of smart contracts [11]. Since smart contracts run on the blockchain, their life cycle is not the same as that of ordinary programs. It is the different life cycle and characteristics of smart contracts that bring new security threats to it.

2.2. Multi-Label Classification

In order to effectively manage information in a structured manner, labeling technology emerges as the times require. As a brief overview of the data information, labels can be quickly indexed and accessed through labels after the data is marked. However, in the face of the trend of data diversification, a single label cannot comprehensively explain and generalize the data information; that is, it cannot better meet the actual needs of multiple tasks. The birth of multi-label technology has become inevitable and reasonable. Multi-labeling of data information can make the original information adapt to more application scenarios. With the development of artificial intelligence, multi-label classification technology has transitioned from classical machine learning to deep learning based on neural networks, and has wide and practical applications in sentiment analysis, information mining, image processing, and other fields.

Du et al. [12] proposed an end-to-end ML-Net model to solve the multi-label classification task of biomedical texts. The model combines a label prediction network with an automatic label number prediction mechanism, using the prediction confidence score of each label, and deep contextual information in the target document to achieve multi-label classification. Alhuzali et al. [13] proposed the SpanEmo model, which uses multi-label sentiment classification as span prediction, which can help sentiment analysis models learn the association between labels and words in sentences, and also introduces a loss function for modeling multiple coexisting emotions. Yogarajan et al. [14] studied the multimorbidity of patients, used a multi-label variant of medical text classification to enhance the prediction of concurrent medical codes, demonstrated new embeddings in health-related text, and dealt with imbalanced compared concentrated variants of embedding models for the multi-label medical text classification problem. We found that multi-label classification methods have achieved more and more results in the field of natural language processing. Smart contract vulnerability detection is a natural language processing problem. Most of the existing research in this field is based on binary classification methods, which can only detect whether the contract has vulnerabilities or not. Therefore, we used a multi-label classification method to detect smart contracts and multiple vulnerability types, so as to

accurately, effectively, and quickly detect vulnerabilities in smart contracts to ensure the security of the blockchain platform.

2.3. Existing Vulnerability Detection Methods

A series of smart contract security vulnerabilities have not only caused huge financial losses to users but also raised serious questions about the fairness and trustworthiness of smart contracts. The security of smart contracts has become a common concern for researchers and developers, and researchers have tried various methods to detect vulnerabilities in smart contracts. The existing smart contract vulnerability detection methods are divided into two main categories, which are traditional vulnerability detection methods and machine learning-based vulnerability detection methods. Among them, traditional vulnerability detection methods include formal verification, symbolic execution, fuzzy testing, and intermediate representation, while machine learning-based vulnerability detection methods include traditional machine learning detection methods and deep learning-based detection methods [15]. Table 1 lists the model frameworks included in each detection method.

Table 1. Existing smart contract vulnerability detection methods.

Method	Model Frameworks
Formal Verification	F* framework [16]; KEVM framework [17]; ZEUS [18]
Symbolic Execution	Oyente [19]; Maian [20]; Securify [21]; Mythril [22]
Fuzzy Testing	ContractFuzzer [23]; Regurad [24]; ILF [25]
Intermediate Representation	Slither [26]; Madmax [27]; Ethir [28]; SmartCheck [29]
Machine Learning	Dynamit [30]; ContractWard [31]
Deep Learning	R-CNN [32]; TSSC [33]; CBGRU [34]

Formal verification transforms the concepts, judgments, and reasoning in a contract into a formal model through formal language. This eliminates ambiguities and implausibilities in the contract and then works with rigorous logic and proof to verify the correctness and security of the function functions in the smart contract. Common formal verification methods include model detection and deductive verification. When symbolic execution is applied in smart contract vulnerability detection, the variable values in the contract are first symbolized, then the instructions in the execution process are interpreted one by one, the execution status is updated and path constraints are collected during the interpretation and execution process, and finally, the exploration of all executable paths in the contract is completed and the corresponding security issues are discovered. Fuzzy testing generates a large number of normal and abnormal test cases from the target application, tries to provide the generated cases to the target application, and monitors the abnormal results in the execution state to discover security issues. Intermediate representation is performed by converting smart contract source code or bytecode into an intermediate representation with high-level semantic representation and then analyzing the intermediate representation of the contract to find security issues. The above traditional detection methods usually rely on fixed semantic rules, are not fully automated, require a second audit by technicians, and have the disadvantages of high false alarm rate and long detection time.

In recent years, machine learning and deep learning have been successfully used in the field of program security, and at the same time, they have promoted new developments in the field of smart contract vulnerability detection. Dynamit is a detection method based on traditional machine learning. It manually extracts features from transactions generated between contracts, adds labels to each transaction manually, and finally inputs a random forest (RF) model for training to obtain high-precision RF classification. This method can only detect whether the contract contains vulnerabilities; it cannot detect the types of vulnerabilities. ContractWard performs N-Gram feature extraction for smart contract opcodes, uses Oyente to detect contracts to add labels and inputs the data into the XGBoost model for training. This method can detect six types of vulnerabilities, and achieve multiple classifications, but the algorithm has large time and space complexity, and memory consumption is too high. R-CNN is a detection model based on a cyclic

neural network (CNN), which converts the bytecode of the vulnerability contract into an RGB image and inputs it into CNN for detection. This method can only detect reentrancy vulnerabilities and is not as effective as it could be. TSSC performs a word embedding operation on the contract at the opcode level to obtain the feature representation of the contract and then inputs the long short-term memory neural network (LSTM) for contract vulnerability detection. This model can detect three vulnerabilities, but ignores the contract before and after the operation code logical relationship between them. CBGRU uses a combination of CNN and GRU models to perform binary detection on three types of vulnerabilities. This method cannot detect which types of vulnerabilities the contract has at the same time, and the CNN model is not good at processing data with sequential relationships. This research uses a model combining Bi-LSTM and an attention mechanism to detect multi-label vulnerabilities in smart contracts. It not only extracts the semantic relationship between the opcodes before and after the contract but also focuses on the important features that have a greater impact on the detection results, to ensure the model detection performance.

3. Related Work

3.1. Smart Contract Vulnerability

Today, various types of smart contract vulnerabilities are emerging, and the reasons for the vulnerabilities are closely related to the functionality of the blockchain platform itself, developer writing, and contract design. Smart contracts are vulnerable to security vulnerabilities for three main reasons. (1) The current smart contract programming language is novel and crude, while contracts are relatively difficult to test in new operating environments. (2) Developers cannot predict the future state and environment of the contract, which makes it easy to write a contract that is vulnerable or susceptible to attack. (3) Unlike traditional programs, smart contract binary code and its state are stored on a tamper-evident blockchain network, and cannot be modified once a smart contract is deployed. Therefore, detecting smart contracts before they are deployed can maximize the security of smart contracts after they are deployed. To achieve smart contract vulnerability detection, this paper conducts research work on the following types of vulnerabilities.

Reentrancy: Reentrancy is a vulnerability arising at the smart contract code level, due to the special fallback mechanism of Solidity smart contracts [35], which allows attackers to reenter the called function to attack the contract before the end of program command execution. Similar to most programming languages, Ethereum smart contracts make cross-contract function calls when handling business logic, but smart contracts often involve sensitive operations such as transferring money. The reentrancy vulnerability arises because an attacker designs malicious attack code in the fallback function to recursively call the contract's transfer function to steal ETH.

Integer overflow/underflow: The types of integer overflow in smart contracts include three types: multiplication overflow, addition overflow, and subtraction overflow. In Ethereum smart contracts, integers are generally specified as fixed size and unsigned integer types, which means that integer variables can only be values within a certain range, beyond which an integer overflow error will occur [36]. Solidity language integer variable steps are generally incremented by 8, supporting types from uint8 to uint256. The uint8 type, for example, has a variable length of 8 bits and supports storing numbers in the range [0,255]. The range of numbers supported for storage is [0, 255]. If an attempt is made to store data in a variable of type uint8 that is larger than this range, the EVM will automatically truncate the high bits, resulting in an integer overflow error. The vulnerability occurs at the smart contract code level.

Transaction order dependency: The order of transaction execution in a blockchain network is determined by miners in the blockchain, and some contracts have strict requirements on the order of transaction execution [37]. For example, user A and user B simultaneously submit transactions T1 and T2 respectively at time t. However, the execution order of T1 and T2 is decided by miners in the block, and if T1 is executed first, the

contract state will change from S to S1 and vice versa, so the final contract state depends on the order of transaction execution selected by miners. If a malicious miner listens to the corresponding contract transaction in the block, he can change the current contract state by submitting a malicious transaction, thus having the opportunity to deploy an attack in advance. This vulnerability is a vulnerability that arises at the blockchain system level.

Timestamp dependency: A security vulnerability introduced by smart contracts that use strict block timestamps in their code to make important control flow decisions. The block timestamp is the timestamp when the block belonging to the current contract call transaction is packed. Block timestamps can be manipulated by miners within a certain range of fetching values [38]. The precise timestamp is used as an important decision parameter in the contract, and while it is somewhat non-defeatable to a normal attacker, a miner-identified attacker can easily circumvent the limitations of the contract's timestamp design by constructing a malicious timestamp with a range of values. This vulnerability also occurs at the blockchain system level.

3.2. Smart Contract Source Code, Bytecode, Opcode

Smart contract development is similar to traditional software development in that a high-level language is used to code it. There are currently more than 40 platforms that support the deployment of smart contracts, all with corresponding contract development languages. Most of these platforms use the Solidity language for development, such as the Ethereum blockchain platform. The programming languages used to develop smart contracts are Vyper, Idris, Rust, etc., in addition to Solidity [39]. The developed smart contract source code needs to be compiled before it can be deployed on the EVM. Program compilation is the translation of program code written in a high-level language into machine code that can be used for machine computation. The binary code used for execution in a smart contract is called bytecode (or EVM code), and EVM completes the execution of the contract by decompiling the bytecode into the corresponding opcode. The Ethereum smart contract is compiled by the compiler, which also generates an application call interface (ABI) for the blockchain to parse out the function selector to implement the contract function calls. We used the Remix tool [40] and Opcode-tool [41] provided by the Ethereum platform to compile source code to bytecode and decompile bytecode to opcode, respectively. Figure 1 shows the relationship between a piece of smart contract source code after compiling and decompiling to generate parts of bytecode and opcode.

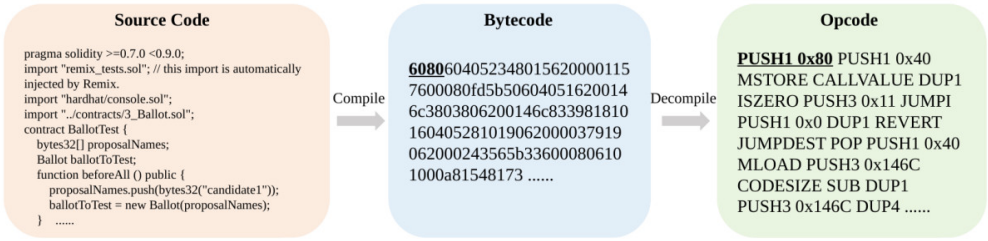

Figure 1. Source code, Bytecode, Opcode conversion relationship.

A bytecode is a string of hexadecimal digit-encoded byte arrays, which are parsed in units of one byte, each of which represents an EVM instruction or an operation data [31]. The bytecode contains a total of three parts: deployment code, runtime code, and auxdata. When EVM creates a contract, it creates the contract account first and then runs the deployment code. After running, the runtime code and auxdata are stored on the blockchain, and finally, the deployment of the contract is completed by associating the storage address of both with the contract account. After deployment, the real execution is the runtime code, so only the runtime code needs to be decompiled to get the contract opcode. Currently, EVM has more than 140 instructions, including arithmetic, comparison, bitwise, cryptographic, stack, memory and storage, and jump instructions, Table 2 summarizes the commonly used Ethereum opcode assembly instructions. By consulting the EVM instruction table [42], we can analyze the bytecode and the opcode in Figure 1: the EVM reads in the hexadecimal

numbers of the bytecode "60 80 60 40 . . . ", and first parses the first byte "60", which corresponds to the "PUSH1" operation in the operation instruction table, because the PUSH1 instruction requires an input parameter, followed by "80", which is the parameter of the PUSH1 instruction, representing the pressing of a byte value into the stack at address 0×80, and then continues to parse the subsequent code.

Table 2. Common assembly instructions.

EVM Code	Opcode Instructions	Description
0×00	STOP	End instructions.
0×01	ADD	Pop the two values at the top of the stack, add and push the result to the top of the stack.
0×02	MUL	Pop the top two values from the stack, multiply and push the result to the top of the stack.
0×03	SUB	Pop the values arg0 and arg1 from the stack in turn, and push the values of arg0-arg1 to the top of the stack.
0×15	ISZERO	Pop the top value of the stack, if the value is 0, push 1 to the stack, otherwise push 0 to the stack.
0×34	CALLVALUE	Get the transfer amount in the transaction.
0×50	POP	Pop the top value from the stack.
0×51	MLOAD	Pop the top of the stack and use the value as the index to load the 32 bytes after the index in memory to the top of the stack.
0×52	MSTORE	Pop the values arg0 and arg1 from the stack in turn, and place arg1 at arg0 in memory.
0×56	JUMP	Pop the top value of the stack and use this value as the destination address of the jump.
0×60	PUSH1	Put a 1-byte value on the top of the stack.
0×80	DUP1	Copy the first value in the current stack to the top of the stack.
0×90	SWAP1	Swap the first value and the second value in the stack.

For the following three reasons, in this study we chose to detect vulnerabilities in smart contracts at the opcode level. (1) The source code is human-written, and the function names in the code are also human-specified. When the function names are changed, the uncertainty in making calls to the functions will affect the detection results, while the source code is long and contains more blank lines and comments, making it difficult to obtain a better feature representation. (2) The bytecode is not readable and it is difficult to directly extract the code syntax structure or the sequence information corresponding to the corresponding function. (3) The opcode is based on more than 140 operation instructions specified by EVM, which can accurately reflect the inner logic of the contract and ensure the reliability of the vulnerability detection model. During the actual dataset collection, we found that more than half of the Ethereum-based smart contracts were not open source, and only bytecode binary files were available, so we decompiled the bytecode into opcode to support model training.

4. Method

To address the problems of poor detection, incomplete automation, and slow detection in traditional detection methods for smart contract vulnerability detection, the goal of this paper was to design a method that can accurately and automatically detect whether a smart contract contains multiple vulnerabilities. The multi-label vulnerability detection framework proposed in this paper consists of two parts, as shown in Figure 2: the data pre-processing module and the vulnerability detection model. The data pre-processing starts with converting the smart contract bytecode into opcodes using the Opcode-tool, followed by converting the opcodes into a word embedding matrix for input to the neural network model using the word embedding model Word2Vec, while we add multi-label vectors to the generated word embedding matrix using Oyente, a static detection tool based on symbolic execution, and the word embedding matrix and multi-label vectors together form the dataset for model training and testing. The training set is fed into the vulnerability detection model for training to obtain a neural network model with high classification

accuracy, and the model is finally validated on a test set. The vulnerability detection model consisted of a Bi-LSTM layer, an attention layer, a fully-connected layer and a sigmoid function to implement multi-label vulnerability detection for smart contracts classification.

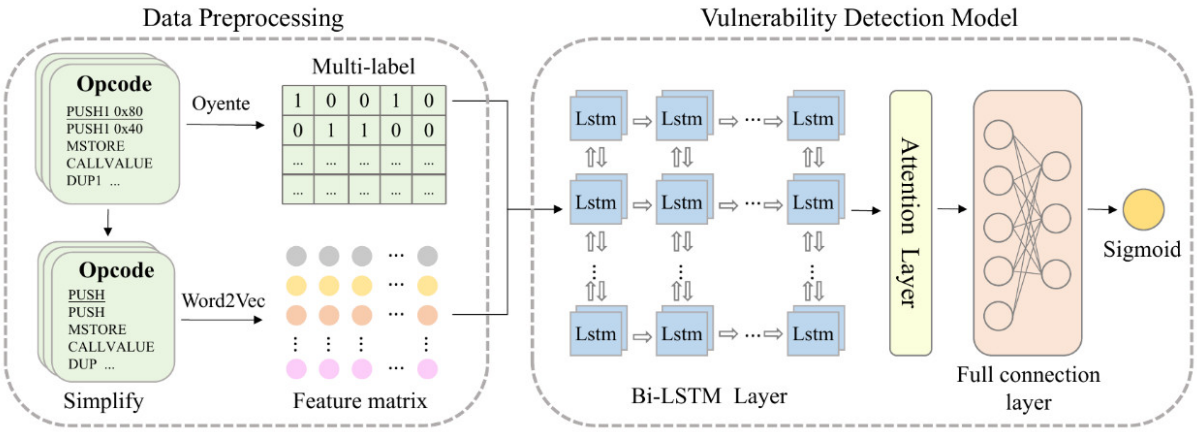

Figure 2. WBL-ATT multi-label vulnerability detection framework.

4.1. Data Preprocessing

In the field of natural language processing (NLP), syntactic analysis, dependency analysis, lexical analysis, and machine translation are indispensable for processing text data. Ordinary text is unstructured data, and unstructured text needs to be "structured" for further analysis and understanding. Structured data makes it easier to extract semantic features in both machine learning and deep learning domains. Word embedding is the main and common approach to solving this problem, which involves embedding a high-dimensional space of all words into a continuous vector space of lower dimensionality, where each word or phrase is mapped to a vector over the real number field [43]. Commonly used word embedding methods are Word2Vec, GloVe, etc. The GloVe is based on traditional statistical methods and does not use neural networks, and the computation is relatively complex, and common Python toolkits do not integrate GloVe. Smart contract opcodes are composed of operation instructions and opcodes, and each opcode can be regarded as a word, which belongs to the textual form of data. To be able to extract and accurately characterize the smart contract vulnerabilities, we chose the Word2Vec model for word embedding operations.

4.1.1. Feature Extraction

Among the more than 140 operation instructions specified by EVM, not all of the opcodes can affect the vulnerability detection results of smart contracts. Opcodes with similar functions can be grouped into the same category to avoid dimensional disasters and reduce computation. For example, by deleting the number 1 after the PUSH1 operation instruction, the partially simplified opcodes were obtained, and are shown in Table 3.

Table 3. Simplified opcode.

Simplified Opcode	Original Opcode
PUSH	PUSH1-PUSH32
LOG	LOG0-LOG4
DUP	DUP1-DUP16
SWAP	SWAP1-SWAP16

The simplified opcodes were vectorized by the Word2Vec model, which is a lightweight neural network consisting of an input layer, a hidden layer, and an output layer. The input layer was the one-hot encoding of the opcode O_i, and according to the simplified set of operational instructions M, each opcode has a number $i \in \{1, \ldots, |M|\}$, and the One-hot

encoding of O_i represents a vector of dimension |M|, where the i-th element value is 1 and the rest of the elements are 0, such as $O_1 = [1, 0, \ldots, 0]^T$. The hidden layer sets a linear activation function, and when the model is trained, the neural network weights are obtained, and the number of weights is the same as the number of nodes in the hidden layer. Only the weight corresponding to position 1 is activated, thus generating a new vector to represent the operation code O_i. Equation (1) represents the feature matrix of the operation code of contract p. The word vector dimension in this study was set to 300, where i represents the i-th opcode in contract p, since the number of opcodes differs for each contract, max denotes the maximum number of contract opcodes in the whole contract dataset, and the vacant position is filled to 0 when $i <$ max.

$$SC_p = \begin{bmatrix} O_{1,1}, & \cdots & ,O_{1,100} \\ O_{2,1}, & \cdots & ,O_{2,100} \\ \vdots & \vdots & \vdots \\ O_{i,1}, & \cdots & ,O_{i,100} \\ \vdots & \vdots & \vdots \\ O_{\max,1}, & \cdots & ,O_{\max,100} \end{bmatrix} \quad (1)$$

4.1.2. Multi-Label

Smart contract multi-label classification is defined as follows: given a dataset $SC = \{(x_p, l_p)\}_{p=1}^{P} = [X, L]$, where $X = [x_1, x_2, \ldots, x_P]^T \in R^{P \times d}, L \in R^{P \times K}$, X refers to the feature space corresponding to the dataset, L refers to the label space of the dataset, N refers to the sample size of the dataset, d refers to the feature vector dimension size, and K refers to the dimension size of the label vector. For the sample (x_p, l_p), x_p refers to the feature vector corresponding to the p-th contract in the dataset; that is, Equation (1), and accordingly, the label vector l_p of the p-th contract consists of K labels, denoted as $l_p = [l_{p1}, l_{p2}, \ldots l_{pK}]$, when it contains the k-th label then it has $l_{pk} = 1$, otherwise $l_{pk} = 1$.

We collected a total of 5700 verified smart contracts from the Ether website and manually added labels to all contracts by the Oyente static detection tool. The model proposed in this paper detects five types of vulnerabilities, which are reentrancy, integer overflow, integer underflow, transaction order dependency, and timestamp dependency, so the above-K value is 5. A contract corresponds to five labels, and the value of each label is 0 or 1. When the value is 0, it means that the contract has no certain vulnerability, and when the value is 1, it means that there is a vulnerability of that type, and each vulnerability label is independent of the other. For example, the label of contract p is $Lable_p = [1, 0, 1, 0, 0]$, which means contract p has reentrancy and integer underflow vulnerability.

4.2. Vulnerability Detection Model

The vulnerability detection model for smart contracts is the core module, which mainly designs and trains specific network models for the vulnerability detection framework, and adjusts the parameters of the models and algorithms in the training process in conjunction with the requirements to obtain the detection model with optimal parameters.

Recurrent neural networks (RNN are commonly used to process sequential data and are widely used in the field of NLP. As the number of network layers increases and the training time lengthens, the original RNN is prone to gradient disappearance and gradient explosion, and to solve this problem, S. Hochreiter and J. Schmidhuber proposed the long short-term memory neural network (LSTM) [44].

LSTM is an improved model based on a RNN, which can efficiently solve multiple learning problems associated with sequential data, better capture the dependencies between longer distance operands in contract sequences, and have better performance in handling long sequence data [45]. The LSTM model is formed by connecting multiple LSTM cells. The cell structure of LSTM is shown in Figure 3. Compared with an RNN, which has only one transmission state, c^t (cell state), LSTM adds a h^t (hidden state), including two

transmission statuses. The input gate, output gate, and forget gate in the LSTM cell jointly realize the functions of long-term and short-term memory, and together with the cell state c^t constitute the LSTM cell. The input gate determines how much input information Z^i is retained in the cell structure, the forget gate determines how much information Z^f should be retained from the hidden state h^{t-1} at the previous moment to the current moment, and the output gate determines how much information Z^o is output from the cell state c^t. Each gate completes the tasks of forgetting and remembering by regulating the cell information. The specific calculation formula is as follows:

$$\tilde{c}^t = \tanh\left(W_c\left[h^{t-1}, x_t\right] + b_c\right) \tag{2}$$

$$Z^i = \sigma\left(W_i\left[h^{t-1}, x_t\right] + b_i\right) \tag{3}$$

$$Z^f = \sigma\left(W_f\left[h^{t-1}, x_t\right] + b_f\right) \tag{4}$$

$$Z^o = \sigma\left(W_o\left[h^{t-1}, x_t\right] + b_o\right) \tag{5}$$

$$c^t = Z^i \times \tilde{c}^t + Z^f \times c^{t-1} \tag{6}$$

$$h^t = Z^o \times \tanh c^t \tag{7}$$

$$y^t = \sigma\left(W_y \times h^t\right) \tag{8}$$

where h^{t-1} and x_t are the transfer of the previous state and the current input respectively, and the transfer h^t and the current output y^t of the next unit are updated through three gating units, W_c, W_i, W_f, W_o are weights, b_c, b_i, b_f, b_o are bias vectors, and σ is the activation function.

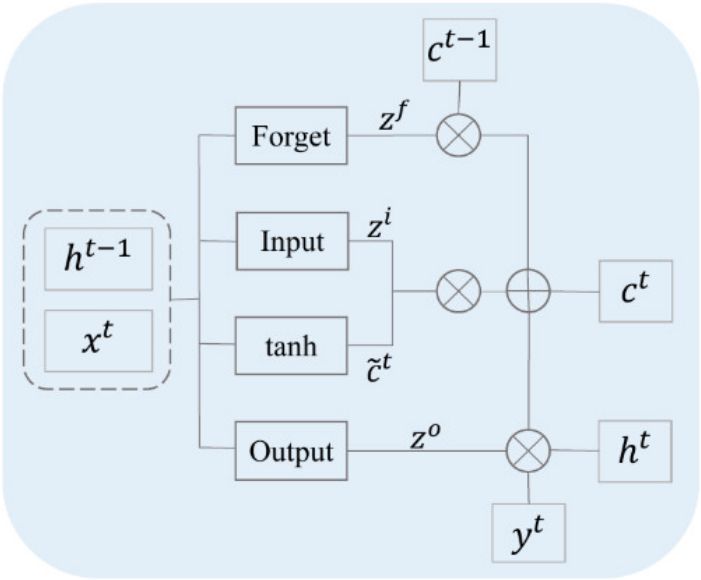

Figure 3. The cell structure of LSTM.

The data processing process of LSTM can be summarized as passing valuable information by forgetting and remembering the information in the transmission state, but it can only realize one-way information transmission, and cannot be analyzed when modeling smart contract datasets. Therefore, for our model we selected a bidirectional long short-term memory neural network (Bi-LSTM) [46] to better capture bidirectional semantic dependencies. The data processing process of one layer of Bi-LSTM is shown in Figure 4.

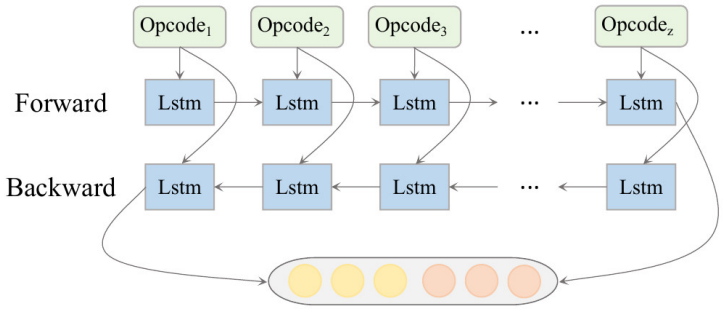

Figure 4. Bi-LSTM data processing process.

Whether it is LSTM or Bi-LSTM, when processing the contract dataset, it can only calculate the opcodes in the forward or reverse order. This mechanism will calculate the time step t depending on the calculation result at the time of $t-1$, and the performance will drop dramatically when processing data sequences that are too long, so we introduced the attention mechanism [47]. As shown in Figure 5, its essence can be described as a query (query) to a series of (key-value) correct mappings. It focuses limited attention on important opcodes and assigns weights to them, thereby saving resources and quickly obtaining the most effective feature information. To highlight the important information implicit in the smart contract opcode, the attention layer first calculates the feature matrix X_c output from the Bi-LSTM layer to obtain its implicit representation H, which is calculated as shown in Equation (9), and then calculates the similarity between H and the random initialization parameter matrix W^T, normalizes the weight coefficients assigned to the input feature vector in the overall semantic scenario using the softmax function to obtain the weights α, and finally multiplies the weight matrix α with the feature matrix X_c to obtain the implicit features Y of the opcode, as shown in the following equation.

$$H = \tanh(W_a \cdot X_c + b_a) \tag{9}$$

$$\alpha = \text{softmax}\left(W^T \cdot H\right) \tag{10}$$

$$Y = X_c \cdot \alpha^T \tag{11}$$

where W_a denotes the attention layer weight matrix and b_a denotes the bias vector of the attention layer.

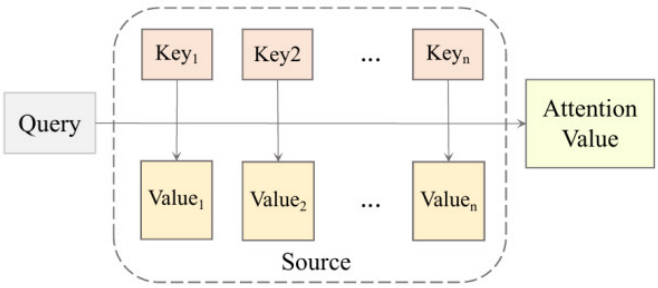

Figure 5. Attention mechanism.

After the data are processed by the Bi-LSTM layer and attention layer, the output is still high-dimensional data compared to the classifiable vulnerability classes, and the output dimensions and parameters of each network layer are shown in Table 4. We added a fully-connected layer to convert the output vector into the dimensionality of a multi-label vector. In contrast to the Bi-LSTM and attention layers, which extract features, the fully-connected layer serves to map the learned distributed feature representation into the sample label space. When processing sequential samples, the fully-connected layer maps the embedding space to the hidden space and then transforms the hidden space to the sample label space to achieve the classification effect, and in the multi-label classification task, the sigmoid

function is used as the activation function of the output layer to model the classes' Bernoulli probability distribution, as shown in Equation (12), and also use binary cross-entropy loss function as the activation function to calculate the loss for each label of a sample and take the average as the final loss to solve the multi-label classification problem, as shown in Equation (13). For the vector output by the fully connected layer, the calculation results of each category are respectively input into the sigmoid function, and the value of the vector is mapped to the probability value between $[0, 1]$. We set the threshold probability to 0.5, and if the probability value is greater than 0.5, it is determined that the sample contains this type of vulnerability, and the output is assumed to be $[0.9, 0.1, 0.2, 0.3, 0.7]$, indicating that the sample contains reentrancy and timestamp dependency vulnerabilities.

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{12}$$

$$\text{Loss} = -\frac{1}{K}\sum_{k=1}^{K}[l_k \cdot \log(\sigma(z)) + (1 - l_k) \cdot \log(1 - \sigma(z))] \tag{13}$$

where z is the output of the fully connected layer, l_k is the true value (0 or 1) corresponding to the k-th class, and $\sigma(z)$ is the output value corresponding to the model; that is, the probability value corresponding to each vulnerability label.

Table 4. The output and parameters of each network layer.

Network Layer	Output Size	Number of Parameters
Bi-LSTM layer	(16,6290,256)	2,021,376
Attention layer	(16,128)	16,521
Fully connected layer	(16,5)	645

5. Experiment

5.1. Dataset and Parameter Settings

By consulting relevant research materials, we found that there is currently no good open-source smart contract vulnerability dataset. Although some datasets have been opened in similar work, the integrity of smart contracts and their data labels cannot be guaranteed. The Ethereum official website obtains the verified smart contract bytecode files to build a reliable experimental dataset, and then uses Oyente, a static detection tool based on symbolic execution, to add multi-label vectors to the contract. There are a total of 5450 contracts with multiple types of vulnerabilities. After verification and deletion of 28 missing value contracts and 132 duplicate contracts, there were 5290 remaining vulnerability contracts. At the same time, in order for the model to better detect vulnerabilities, we also constructed 1000 contract datasets that did not contain vulnerabilities for experiments, and the ratio of training set to test set was 8:2. The number of contracts containing each vulnerability in the final dataset is shown in Table 5.

Table 5. Number of smart contracts.

Vulnerability Category	Reentrancy	Integer Overflow	Integer Underflow	Transaction Order Dependency	Timestamp Dependency	None
Amount	826	3711	3194	2594	1228	1000
Total amount	6290					

The experiments were run in the official docker container of PyTorch and trained by using Nvidia RTX3070 graphics card. Since the parameter settings of different models are different, we used the same parameters as much as possible to ensure a fair evaluation of each model. The hidden layer dimension of RNN, LSTM, and BiLSTM was set to 128, and their stacking layers were set to 5 layers. The learning rate was uniformly set to 0.001, the epoch was set to 100, the optimizer used the Adam optimizer, the batch size was set to 64,

the dimension of the input word vector was 300, and the maximum length of the word vector was 1000. The loss function used binary cross-entropy.

5.2. Evaluation Indicators

The evaluation indicators commonly used in the experiment included: recall, precision, F1-score, and accuracy. Based on the confusion matrix in Table 6, the calculation of each indicator was as the following equation shows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{14}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{15}$$

$$\text{F1} - \text{score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{16}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{17}$$

where TN (true negative) represents the number of samples whose predicted value and true value are both negative, FN (false negative) represents the number of samples whose predicted value is negative and the true value is positive. TP (true positive) represents the predicted value and the number of samples whose true values are all positive classes, and FP (false positive) represents the number of samples whose predicted values are positive classes and the true values are negative classes. The experiment in this paper was multi-label classification, so we chose micro-recall rate (micro-R), micro-precision rate (micro-P), and micro-F1-Score (micro-F1) to comprehensively evaluate the multi-vulnerability detection effect of the model. The calculation equation was as follows:

$$\text{micro} - \text{R} = \frac{\overline{\text{TP}}}{\overline{\text{TP}} \times \overline{\text{FN}}} \tag{18}$$

$$\text{micro} - \text{P} = \frac{\overline{\text{TP}}}{\overline{\text{TP}} \times \overline{\text{FP}}} \tag{19}$$

$$\text{micro} - \text{F1} = \frac{2 \times (\text{micro} - \text{R}) \times (\text{micro} - \text{P})}{(\text{micro} - \text{R}) + (\text{micro} - \text{P})} \tag{20}$$

Table 6. Confusion matrix.

True \ **Predicate**	**0**	**1**
0	TN	FP
1	FN	TP

5.3. Experimental Results

To verify the effectiveness of the model designed in this study, RNN, LSTM, LSTM-ATT, Bi-LSTM, and TextCNN [48] were used to compare with our model. The evaluation results of the six models on the experimental dataset are shown in Table 7. It can be seen that the multiple multi-label evaluation indicators of our proposed model are higher than those of the comparison model, which reflects the effectiveness of the WBL-ATT model in the field of multi-label detection of smart contract vulnerabilities. The higher micro-R value of RNN and the lower micro-P value is because the RNN model suffers from severe overfitting, and the model is more inclined to predict the vulnerability of the sample as the category of vulnerabilities in the training set, while other models micro-P values are higher than RNN, indicating that it would be wise for us to introduce sequential model LSTM to process opcodes. By comparing the indicators of LSTM and Bi-LSTM, it is proved that extracting reverse semantic information between contract opcodes is conducive to detecting

contract vulnerabilities. In addition, we also found that the indicators of LSTM-ATT were greatly improved compared to Bi-LSTM. Combined with the analysis of the experimental process, it is concluded that the addition of an attention mechanism can make the model balance features with a small number of samples and focus on important feature data, which has a greater impact on vulnerability detection results than Bi-LSTM. TextCNN is an improved CNN model for handling NPL tasks, but the network structure of this model is relatively simple, the feature extraction ability is limited, and the effect performance is general. Its metrics are much lower than those of our model.

Table 7. Model evaluation results.

Model	Micro-R (%)	Micro-P (%)	Micro-F1 (%)	Accuracy (%)
RNN	83.26	70.73	76.49	73.90
LSTM	67.33	76.28	71.53	74.26
Bi-LSTM	70.71	79.76	74.97	76.24
LSTM-ATT	79.68	81.70	80.72	80.71
TextCNN	67.71	71.85	72.11	71.03
WBL-ATT	87.25	87.71	87.51	88.12

Figure 6 shows the accuracy values of the six models for each of the five vulnerabilities. The accuracy value is the ratio of the number of correctly predicted samples to the total number of samples, and it can be found that the accuracy of WBL-ATT that the model we proposed for the detection of all five vulnerabilities exceeded that of the other models, with TOD and timestamp dependency being much higher than in the other models. Due to the small number of samples containing reentrancy and timestamp dependency vulnerabilities in the dataset, RNN and LSTM performed well, but in fact, because these two models had overfitting phenomenon, the models tended to predict the samples as not containing these two vulnerabilities. Comparing the LSTM and Bi-LSTM models, we found that the bidirectional LSTM model performed better than the unidirectional LSTM model in all vulnerabilities. The TextCNN performed poorly except for on the integer underflow vulnerability, which verifies that the TextCNN model is simpler and therefore weak in feature extraction.

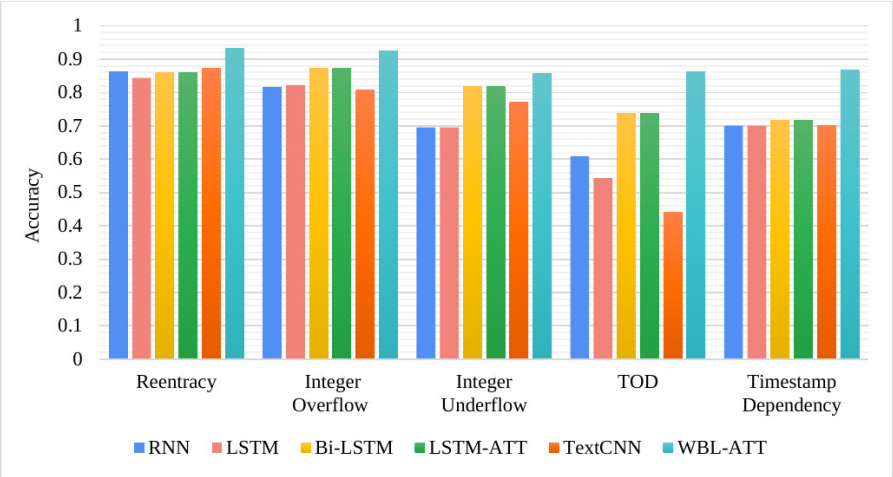

Figure 6. Accuracy of five vulnerabilities detection.

Table 8 shows the precision, recall, and F1-score of the five vulnerabilities detected by the six models respectively. The precision value can measure the number of correct predictions in the samples predicted by the model; the recall value is used to indicate how many of the actual samples were correctly predicted. The F1 value is an indicator used in statistics to measure the accuracy of the binary classification model. It takes into account the precision and recall of the classification model and can be regarded as a weighted

average of the precision and recall of the model. From the table, it can be found that RNN, LSTM, and Bi-LSTM all had serious overfitting. This is because the number of samples of integer overflow, integer underflow and TOD in the training samples was much larger than the number of reentrancy and timestamp dependency samples. The model trained on unbalanced data was more inclined to include all three types of vulnerabilities, so the recall values of these three types of vulnerabilities were all 1, while the recall value of the other two types of vulnerabilities with fewer samples was 0. The LSTM-ATT and WBL-ATT with the added attention mechanism could reasonably balance different vulnerabilities, reflecting that the attention mechanism can effectively enhance the generalization ability of the model. Although the WBL-ATT model also had a low accuracy of vulnerability classification with a small number of samples, the precision value on reentrancy was only 72.26%, which was lower than those of LSTM-ATT and TextCNN, but timestamp dependency reached 78.97%, much higher than the values of LSTM-ATT and TextCNN, and among the three samples with a large number of vulnerabilities, integer overflow, integer underflow, TOD, and WBL-ATT performed well, and the precision value was greater than that of TextCNN and LSTM-ATT. In general, the WBL-ATT model had good generalization performance and performed well on samples with a large number of vulnerabilities.

Table 8. Precision, recall and F1-score of five vulnerabilities detection.

Model	Indicator (%)	Reentrancy	Integer Overflow	Integer Underflow	TOD	Timestamp Dependency
RNN	Precision	0	79.32	69.43	60.06	0
	Recall	0	100.00	100.00	100.00	0
	F1-score	0	88.63	82.03	75.70	0
LSTM	Precision	0	80.70	69.54	79.80	0
	Recall	0	100.00	100.00	33.33	0
	F1-score	0	90.16	81.35	47.05	0
Bi-LSTM	Precision	0	94.68	71.82	73.60	0
	Recall	0	82.60	92.86	76.60	0
	F1-score	0	85.60	80.89	75.02	0
LSTM-ATT	Precision	90.90	89.53	84.82	75.00	55.56
	Recall	24.00	94.65	88.78	84.92	25.42
	F1-score	35.29	91.49	86.32	78.68	34.83
TextCNN	Precision	80.00	94.10	83.44	77.16	68.18
	Recall	16.00	91.92	83.21	81.66	50.84
	F1-score	26.67	93.36	82.32	79.35	58.25
WBL-ATT	Precision	72.26	95.67	89.89	87.56	78.97
	Recall	72.00	95.66	88.76	86.83	77.63
	F1-score	73.24	95.99	89.23	87.19	78.59

In order to verify the reliability of our proposed Word2Vec model for processing opcode data, we used the NN Embedding layer that comes with Pytorch to process word vectors to convert opcodes, and input the WBL-ATT model together with our proposed Word2Vec feature matrix. The results of the four evaluation indicators are shown in Figure 7. It can be seen that the data extracted by the Word2Vec model was more beneficial to the subsequent detection results, but it could not achieve end-to-end model training, which provides our new future research direction.

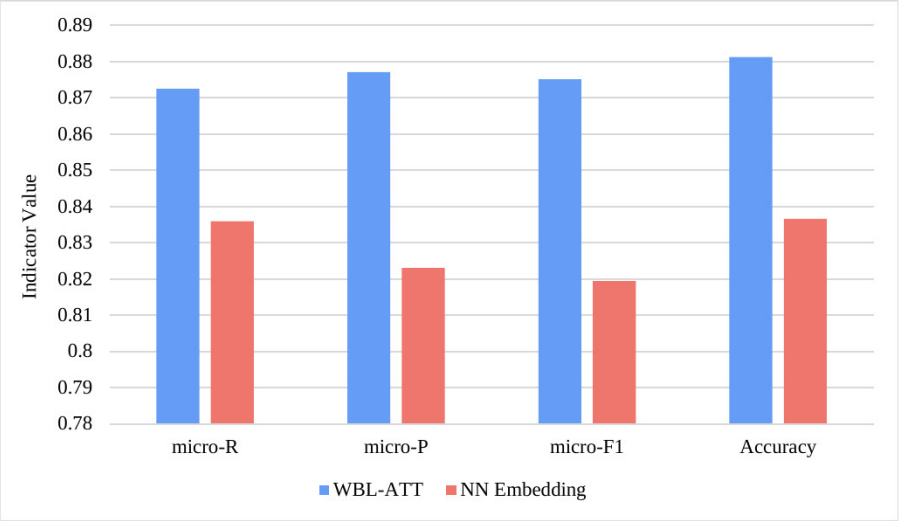

Figure 7. Word embedding model comparison.

6. Conclusions and Future Work

Smart contracts have expanded the application scenarios of blockchain, and the number of contracts on various blockchain platforms has increased exponentially. However, frequent smart contract vulnerability attacks not only cause heavy losses to users but also bring into question the reliability of blockchain platforms. In our work, we proposed an approach combining Bi-LSTM and attention mechanisms to implement multi-label vulnerability detection for smart contracts. For the Ethereum smart contract dataset, the bytecode was parsed to obtain the corresponding opcode, and the Word2Vec word embedding model was used to convert the opcode into a word embedding matrix for the neural network input, and then the Bi-LSTM was used to extract the semantic information between the front and back of the opcode, while the attention mechanism was added to assign weights to the important features, and finally the fully connected layer and sigmoid function were used to achieve multi-label classification. We tested our proposed multi-label vulnerability classification detection model with several vulnerability classification detection models on a self-made multi-label vulnerability dataset, and the results showed that our proposed multi-label vulnerability classification detection model outperforms other models, indicating that our proposed approach performs well in handling the multi-label vulnerability detection task of smart contracts.

Although this research has contributed to the field of smart contract vulnerability detection, there are still limitations to a certain extent: the relationship between the code in a smart contract and different vulnerabilities is a key issue for vulnerability detection in smart contracts and requires further digging. The following aspects are planned to be optimized in future research work: (1) Locate the vulnerability of the contract that has detected the vulnerability, and find out which operation instructions cause the vulnerability of the contract. (2) Our method is only for vulnerability detection of smart contracts on the Ethereum platform, and the next step is to extend it to other blockchain platforms (such as Fabric, ESO) to achieve broader contract vulnerability detection. (3) With the advancement of technology in the NLP field, researchers have proposed more new frameworks, and we will try to use end-to-end technology to process opcodes to achieve vulnerability classification.

Author Contributions: Conceptualization S.Q. and Y.H.; methodology, H.N. and M.C.; software, H.N. and M.C.; validation, M.C.; writing—original draft preparation, H.N.; writing—review and editing, S.Q. and H.N.; visualization, H.N. and M.C.; supervision, S.Q. and Y.H.; project administration, S.Q. and Y.H.; funding acquisition, S.Q. and Y.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by major public welfare projects in Henan Province Research on key technologies of blockchain system security, grant number 201300210200.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Swan, M. Blockchain: Blueprint for a New Economy. 2015. Available online: http://www.abbeys.com.au/book/bitcoin-a-blueprint-for-a-new-world-currency.do (accessed on 7 August 2022).
2. Yuan, Y.; Wang, F.Y. Blockchain: The State of the Art and Future Trends. *Acta Autom. Sin.* **2016**, *42*, 481–494. [CrossRef]
3. Wright, D.C.S. Bitcoin: A Peer-to-Peer Electronic Cash System. Rochester, NY, USA. 21 August 2008. Available online: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3440802 (accessed on 30 August 2022).
4. Szabo, N. Smart contracts: Building blocks for digital markets. *EXTROPY J. Transhumanist Thought* **1996**, *18*, 28.
5. Shao, Q.F.; Jin, C.Q.; Zhang, Z.; Qian, W.N.; Zhou, A.Y. Blockchain: Architecture and Research Progress. *Chin. J. Comput.* **2018**, *41*, 969–988. [CrossRef]
6. Mehar, M.; Shier, C.; Giambattista, A.; Gong, A.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *J. Cases Inf. Technol.* **2017**, *21*, 19–32. [CrossRef]
7. Huang, J.; Zhou, K.; Xiong, A.; Li, D. Smart Contract Vulnerability Detection Model Based on Multi-Task Learning. *Sensors* **2022**, *22*, 1829. [CrossRef] [PubMed]
8. He, H.; Yan, A.; Chen, Z. Survey of Smart Contract Technology and Application Based on Blockchain. *J. Comput. Res. Dev.* **2018**, *55*, 2452–2466. [CrossRef]
9. Lin, S.-Y.; Zhang, L.; Li, J.; Ji, L.; Sun, Y. A survey of application research based on blockchain smart contract. *Wirel. Netw.* **2022**, *28*, 635–690. [CrossRef]
10. Jani, S. *Smart Contracts: Building Blocks for Digital Transformation*; Indira Gandhi National Open University: New Delhi, India, 2020.
11. Bin, L.W.-Z.C.; Bing, Q.; Yi, S.N.-W.G.; Yun, S. A review of the application of blockchain technology in power auxiliary Services. *Power Syst. Technol.* **2017**, *41*, 736–744.
12. Du, J.; Chen, Q.; Peng, Y.; Xiang, Y.; Tao, C.; Lu, Z. ML-Net: Multi-label classification of biomedical texts with deep neural networks. *J. Am. Med. Inform. Assoc.* **2019**, *26*, 1279–1285. [CrossRef]
13. Alhuzali, H.; Ananiadou, S. SpanEmo: Casting Multi-label Emotion Classification as Span-prediction. *arXiv* **2021**, arXiv:2101.10038.
14. Yogarajan, V.; Montiel, J.; Smith, T.; Pfahringer, B. Seeing the Whole Patient: Using Multi-Label Medical Text Classification Techniques to Enhance Predictions of Medical Codes. *arXiv* **2020**, arXiv:2004.00430.
15. Huang, J.; Han, S.; You, W.; Shi, W.; Liang, B.; Wu, J.; Wu, Y. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 2144–2156. [CrossRef]
16. Grishchenko, I.; Maffei, M.; Schneidewind, C. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust*; Springer: Cham, Switzerland, 2018; pp. 243–269. [CrossRef]
17. Hildenbrandt, E.; Saxena, M.; Rodrigues, N.; Zhu, X.; Daian, P.; Guth, D.; Moore, B.; Park, D.; Zhang, Y.; Stefanescu, A. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF), Oxford, UK, 9–12 July 2018; pp. 204–217. [CrossRef]
18. Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. ZEUS: Analyzing Safety of Smart Contracts. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
19. Badruddoja, S.; Dantu, R.; He, Y.; Upadhayay, K.; Thompson, M. Making Smart Contracts Smarter. In Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Sydney, Australia, 3–6 May 2021; pp. 1–3. [CrossRef]
20. Nikolic, I.; Kolluri, A.; Sergey, I.; Saxena, P.; Hobor, A. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018.
21. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018.
22. Zhang, P.; Xiao, F.; Luo, X. A Framework and DataSet for Bugs in Ethereum Smart Contracts. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 28 September–2 October 2020.
23. Jiang, B.; Liu, Y.; Chan, W.K. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 259–269. [CrossRef]

24. Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; Roscoe, B. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May–3 June 2018; pp. 65–68.
25. He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 531–548. [CrossRef]
26. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 27 May 2019; pp. 8–15. [CrossRef]
27. Grech, N.; Kong, M.; Jurisevic, A.; Brent, L.; Scholz, B.; Smaragdakis, Y. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* **2018**, *2*, 116:1–116:27. [CrossRef]
28. Albert, E.; Gordillo, P.; Livshits, B.; Rubio, A.; Sergey, I. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis*; Springer: Cham, Switzerland, 2018; pp. 513–520. [CrossRef]
29. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. SmartCheck: Static Analysis of Ethereum Smart Contracts. In Proceedings of the 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden, 27 May 2018; pp. 9–16.
30. Eshghie, M.; Artho, C.; Gurov, D. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. *arXiv* **2021**, arXiv:2102.07420.
31. Wang, W.; Song, J.; Xu, G.; Li, Y.; Wang, H.; Su, C. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 1133–1144. [CrossRef]
32. Huang, T.T. Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks. *arXiv* **2018**, arXiv:1807.01868. [CrossRef]
33. Tann, W.J.-W.; Han, X.; Gupta, S.; Ong, Y.-S. Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Vulnerabilities. *arXiv* **2018**, arXiv:1811.06632.
34. Zhang, L.; Chen, W.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. CBGRU: A Detection Method of Smart Contract Vulnerability Based on a Hybrid Model. *Sensors* **2022**, *22*, 3577. [CrossRef]
35. Grossman, S.; Abraham, I.; Golan-Gueta, G.; Michalevsky, Y.; Rinetzky, N.; Sagiv, M.; Zohar, Y. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.* **2017**, *2*, 48. [CrossRef]
36. Lai, E.; Luo, W. Static Analysis of Integer Overflow of Smart Contracts in Ethereum. In Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, 10–12 January 2020; pp. 110–115. [CrossRef]
37. Ni, Y.; Zhang, C.; Yin, T. A survey of smart contract vulnerability research. *J. Cyber Secur.* **2020**, *5*, 78–99.
38. Atzei, N.; Bartoletti, M.; Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 164–186. [CrossRef]
39. Macrinici, D.; Cartofeanu, C.; Gao, S. Smart contract applications within blockchain technology: A systematic mapping study. *Telem. Inform.* **2018**, *35*, 2337–2354. [CrossRef]
40. Remix—Ethereum IDE. Available online: https://remix.ethereum.org (accessed on 30 August 2022).
41. Bytecode to Opcode Disassembler. Etherscan. Available online: http://etherscan.io/opcode-tool (accessed on 30 August 2022).
42. Ethereum Virtual Machine Opcodes. Available online: https://ethervm.io/ (accessed on 30 August 2022).
43. Kuang, S.; Davison, B.D. Learning class-specific word embeddings. *J. Supercomput.* **2020**, *76*, 8265–8292. [CrossRef]
44. Hochreiter, S.; Schmidhuber, J. Long Short-term Memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
45. Ma, S.; Cui, J.; Chen, C.-L.; Xiao, W.; Liu, L. An improved bi-LSTM EEG emotion recognition algorithm. *J. Netw. Intell.* **2022**, *7*, 623–639.
46. Huang, Z.; Xu, W.; Yu, K. Bidirectional LSTM-CRF Models for Sequence Tagging. *arXiv* **2015**, arXiv:1508.01991.
47. Woo, S.; Park, J.; Lee, J.-Y.; Kweon, I.S. CBAM: Convolutional Block Attention Module. In *Computer Vision—ECCV 2018*; Springer: Cham, Switzerland, 2018; pp. 3–19. [CrossRef]
48. Kim, Y. Convolutional Neural Networks for Sentence Classification. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, 26–28 August 2014. [CrossRef]