electronics

MDPI

Article

Ethereum Smart Contract Vulnerability Detection Model Based on Triplet Loss and BiLSTM

Meiying Wang *, Zheyu Xie, Xuefan Wen, Jianmin Li and Kuanjiu Zhou

School of Software, Dalian University of Technology, Dalian 116620, China
* Correspondence: 22117023@mail.dlut.edu.cn

Abstract: The wide application of Ethereum smart contracts in the Internet of Things, finance, medical, and other fields is associated with security challenges. Traditional detection methods detect vulnerabilities by stacking hard rules, which are associated with the bottleneck of a high false-positive rate and low detection efficiency. To make up for the shortcomings of traditional methods, existing deep learning methods improve model performance by combining multiple models, resulting in complex structures. From the perspective of optimizing the model feature space, this study proposes a vulnerability detection scheme for Ethereum smart contracts based on metric learning and a bidirectional long short-term memory (BiLSTM) network. First, the source code of the Ethereum contract is preprocessed, and the word vector representation is used to extract features. Secondly, the representation is combined with metric learning and the BiLSTM model to optimize the feature space and realize the cohesion of similar contracts and the discreteness of heterogeneous contracts, improving the detection accuracy. In addition, an attention mechanism is introduced to screen key vulnerability features to enhance detection observability. The proposed method was evaluated on a large-scale dataset containing four types of vulnerabilities: arithmetic vulnerabilities, re-entrancy vulnerabilities, unchecked calls, and inconsistent access controls. The results show that the proposed scheme exhibits excellent detection performance. The accuracy rates reached 88.31%, 93.25%, 91.85%, and 90.59%, respectively.

Keywords: smart contract; vulnerability detection; triplet loss; attention mechanism

Citation: Wang, M.; Xie, Z.; Wen, X.; Li, J.; Zhou, F. Ethereum Smart Contract Vulnerability Detection Model Based on Triplet Loss and BiLSTM. *Electronics* **2023**, *12*, 2327. https://doi.org/10.3390/electronics12102327

Received: 10 April 2023
Revised: 6 May 2023
Accepted: 9 May 2023
Published: 22 May 2023

1. Introduction

A smart contract is a transaction protocol that runs on the upper layer of the blockchain. Relying on the unforgeable, tamper-proof, traceable, and decentralized features of the blockchain, smart contracts provide complete transaction services for applications in different fields [1]. The Ethereum platform is currently one of the most widely used blockchain platforms for smart contracts. It has launched a Turing-complete smart contract programming language, Solidity, which is widely used in financial services, power supply systems, the Internet of Things, and the medical field [2]. In 2015, Fabian Vogelsteller proposed the ERC-20 token standard to provide standardized specifications for the issuance and exchange of tokens on the Ethereum blockchain [3]. In 2017, Abdullah Albeyatti and technical expert Mo Tayeb designed the distributed medical data platform Medicalchain, which realized safe and efficient medical data management and drug traceability while protecting patient privacy [4]. In the same year, PowerLedger and Grid+ were proposed for energy trading and power market and grid management, providing consumers with transparent and efficient power market services [5]. In 2022, Goudarzi et al. analyzed and discussed the application of Internet of Things technology in smart grids, pointing out that in the future, blockchain technology needs to be used to solve the challenges faced by smart grids in terms of data privacy protection, energy transaction security, and energy consumption management [6]. In the same year, Waseem et al. proposed a blockchain-based smart grid architecture to achieve decentralized, credible, and secure energy management; however, this architecture has limitations in terms of big data performance and supervision [7]. Aiming at the IoT security issues under the high concurrent traffic of large-scale IoT devices,

in 2021, Kumar et al. proposed a distributed architecture of multiple fog node collaboration to detect distributed denial-of-service attacks in smart contracts, blockchains, and IoT systems; however, the proposed architecture requires considerable computing resources [8]. In 2022, Zhou et al. analyzed the security vulnerabilities of Ethereum smart contracts and used a support vector machine (SVM) classifier to classify contracts [9]. In the same year, Gupta et al. proposed a smart contract feature extraction method based on a convolutional neural network (CNN) and a long short-term memory (LSTM) network combined with the random forest algorithm to classify contracts to detect malicious smart contracts in the Internet of Things environment [10]. Recent research has initially explored the combined application of the blockchain and the Internet of Things, but challenges remain in terms of performance and security that need to be further studied and developed.

Compared with traditional applications, blockchain-based smart contracts have several significant differences. First of all, the contract code is relatively small in size and simple in logic and can be implemented through simple if…else…statements. When the conditions are met, the contract is executed automatically without human intervention. Second, once the contract is deployed on the chain, the contract and the data it generates cannot be modified again. Therefore, if a smart contract is deployed on the blockchain in the presence of vulnerabilities, it may cause huge economic losses. For example, in 2016, the DAO lost ETH 3.6 million due to a re-entrancy vulnerability, resulting in a loss of USD 70 million [11]. In 2018, the BEC campaign lost more than USD 900 million due to integer overflow vulnerabilities [12].

To avoid the abovementioned security crises caused by smart contract vulnerabilities, researchers have proposed various detection methods for smart contracts based on existing code vulnerability detection technologies, such as symbolic execution, fuzzy testing, and formal verification, to detect known vulnerabilities [13–15]. With the rapid increase in the number of smart contracts, traditional vulnerability detection techniques can no longer meet the audit needs of a large number of smart contracts in a short period of time. Therefore, deep learning technology has gradually become a more effective means of smart contract vulnerability detection. Deep learning technology trains models by extracting features from large-scale data, and its feature representation ability has an important impact on the accuracy of vulnerability detection [16]. At present, most researchers use multimodel parallel or serial feature extraction to capture the most representative key features, but no scholars have optimized the feature learning ability of deep learning models from the perspective of vector space representation. In addition, existing deep learning schemes usually use the bytecode of smart contracts as input data, which is not conducive to developers' code writing and debugging. Therefore, it is necessary to develop more source-code-based deep learning solutions to improve the efficiency and accuracy of vulnerability detection.

To address the abovementioned issues, we propose a vulnerability detection scheme for Ethereum smart contracts based on metric learning triplet loss and bidirectional a long short-term memory (BiLSTM) network model. Our main contributions are as follows:

(1) A novel vulnerability detection scheme is proposed from the perspective of feature representation space optimization. In contrast to existing methods that improve the feature learning ability by combining multiple models, we optimize and evaluate the features extracted by the model based on metric learning, making smart contracts of the same category more cohesive and smart contracts of different categories more discrete, improving the accuracy of vulnerability detection.
(2) The proposed model enhances the interpretability of contract vulnerability detection. Our approach involves using the source code of Ethereum smart contracts as input data, which is then subjected to word vectorization and an attention mechanism. This process allows us to identify the critical features associated with vulnerabilities, aiding in pinpointing the root cause of the issue.
(3) We construct a large-scale dataset of smart contracts. We collected 165,000 verified source codes of smart contracts and used a variety of vulnerability detection tools to assign vulnerability labels to provide more comprehensive data support for detection.

(4) The proposed model improves detection accuracy. Experiments prove that compared with traditional methods and other deep learning models, the proposed scheme can better extract vectorized features of smart contracts and effectively improve the accuracy of vulnerability detection.

The rest of this paper is organized as follows. Section 2 introduces important and common types of vulnerabilities in smart contracts and related research on existing smart contract vulnerability detection techniques. Section 3 details the framework of our smart contract vulnerability detection scheme based on triplet loss and the bidirectional long short-term memory network model. Section 4 presents the experimental procedure of this scheme and a performance comparison with other schemes. Section 5 summarizes the work reported in this paper and elaborates on the direction of further research in the future.

2. Background

To accurately detect the vulnerabilities in smart contracts, we conducted an in-depth study on the common vulnerabilities in current contracts and existing smart contract vulnerability detection technologies.

2.1. Smart Contract Vulnerabilities

Once a smart contract is deployed on the chain, the operations generated when an unsafe contract is attacked cannot be reversed. There are many possible reasons for contract insecurity, including functional limitations caused by the inherent properties of the blockchain platform, logic errors caused by developers' negligence in the coding process, and the order of contract business processes [17]. We modeled and analyzed the following four typical vulnerability types based on the datasets we collected.

Arithmetic vulnerabilities: Arithmetic vulnerabilities in smart contracts mainly involve integer overflow and underflow issues. These problems usually occur during integer operations in contracts, which may lead to unexpected results and potential security issues [18]. Integer overflow means that when an integer variable increases beyond its maximum allowed value, it restarts calculations from the minimum value. For example, in Solidity, the maximum value allowed for an integer of type uint256 is $2^{256} - 1$. When this integer variable is incremented by 1, it becomes 0, as shown in Figure 1. Correspondingly, integer underflow means that when an integer variable decreases below its allowed minimum value, it restarts calculations from the maximum value. For example, in Solidity, the minimum value allowed for an integer of type uint256 is 0. When that integer variable is decremented by 1, it becomes $2^{256} - 1$, which is integer underflow. Arithmetic bugs can lead to lost funds, incorrect transfers, and other security issues.

```
1  pragma solidity ^0.8.0;
2
3  contract Crowdsale {
4      mapping(address => uint256) public balances;
5      uint256 public totalRaised;
6
7      function contribute() public payable {
8          require(msg.value > 0, "Contribution amount must be greater than zero");
9          balances[msg.sender] += msg.value;
10         totalRaised += msg.value;
11     }
12
13     function withdraw() public {
14         uint256 amount = balances[msg.sender];
15         require(amount > 0, "No funds available to withdraw");
16         balances[msg.sender] = 0;
17         payable(msg.sender).transfer(amount);
18     }
19 }
```

Figure 1. Example of arithmetic vulnerability.

In the contribute function in line 7 of the crowdfunding contract example presented in Figure 1, both `balances[msg.sender]` and `totalRaised` may have integer overflow vulnerabilities.

Re-entrancy Vulnerabilities: Re-entrancy vulnerabilities are caused by a contract calling an external contract before completing a state update [19]. In this case, the called external contract may call the original contract again, causing the state of the original contract to be changed unexpectedly.

The `withdraw()` function in Figure 2 uses the low-level call function on line 12 to send ETH back to the user before updating the balance on line 14. However, transferring funds before updating user balances leaves room for re-entrancy attacks. An attacker can create a malicious contract, as shown in Figure 3, which calls the `withdraw()` function again when receiving ETH, resulting in repeated payments.

```
1   pragma solidity ^0.6.0;
2
3   contract VulnerableBank {
4       mapping(address => uint256) public balances;
5
6       function deposit() public payable {
7           balances[msg.sender] += msg.value;
8       }
9
10      function withdraw() public {
11          uint256 amount = balances[msg.sender];
12          (bool success, ) = msg.sender.call{value: amount}("");
13          require(success, "Withdrawal failed");
14          balances[msg.sender] = 0;
15      }
16  }
```

Figure 2. Example of re-entrancy vulnerability.

```
1   pragma solidity ^0.6.0;
2
3   contract MaliciousContract {
4       VulnerableBank public vulnerableBank;
5
6       constructor(VulnerableBank _vulnerableBank) public {
7           vulnerableBank = _vulnerableBank;
8       }
9
10      function attack() public payable {
11          vulnerableBank.deposit{value: msg.value}();
12          vulnerableBank.withdraw();
13      }
14
15      receive() external payable {
16          if (address(vulnerableBank).balance > 1 ether) {
17              vulnerableBank.withdraw();
18          }
19      }
20  }
```

Figure 3. Example of a re-entrancy vulnerability attack contract.

In this malicious contract, the `attack()` function on line 10 first deposits to VulnerableBank, then calls the `withdraw()` function. When ETH is sent to the malicious contract, its `receive()` function is triggered, and the `withdraw()` function is called again, causing the original contract to pay multiple times.

Unchecked calls: Unchecked calls are mainly caused by contract developers not correctly checking the call results when calling other contracts or sending ETH [20]. When the call fails, it may cause unexpected behavior or other security issues in the

contract. In Ethereum's Solidity smart contract, contracts can call each other through low-level functions such as call, delegatecall, and send. However, these low-level functions do not automatically throw exceptions but instead return a Boolean indicating whether the call was successful or not. If the developer fails to check the return value when calling these functions, it may lead to an unchecked call vulnerability.

In Figure 4, the `transfer` function sends ETH to the target address via the `call.value` function. However, it does not check the result of the call, which could cause other problems in the contract if the call fails.

```
1  pragma solidity ^0.6.0;
2  contract UncheckedCall {
3      function transfer(address to, uint amount) public payable {
4          bool success = to.call.value(amount)("");
5          // Unchecked Call Result
6      }
7  }
```

Figure 4. Example of unchecked call vulnerability.

Inconsistent access control: Inconsistent access control is mainly due to the negligence or inexperience of contract developers in designing and implementing access control strategies, resulting in improper smart contract permission control, which may allow unauthorized users to access or modify sensitive information and functions of the contract [21]. Access control issues can lead to security risks such as loss of funds, contract tampering, data leakage, etc.

The `withdrawAllFunds()` function on line 20 in Figure 5 does not implement the access control policy correctly, and anyone can withdraw all funds after the contract is unlocked. This could allow unauthorized users to perform sensitive operations and result in loss of funds. To fix this, we need to add a check on the `msg.sender` function to make sure only the owner can do this.

```
1  pragma solidity ^0.6.0;
2
3  contract InconsistentAccessControl {
4      address public owner;
5      uint256 public lockedUntil;
6
7      constructor() public {
8          owner = msg.sender;
9          lockedUntil = block.timestamp + 1 days;
10     }
11
12     function changeOwner(address newOwner) public {
13         require(msg.sender == owner, "Only the owner can change ownership");
14         owner = newOwner;
15     }
16
17     function unlock() public {
18         require(msg.sender == owner, "Only the owner can unlock the contract");
19         lockedUntil = block.timestamp + 1 days;
20     }
21
22     function withdrawAllFunds() public {
23         require(block.timestamp > lockedUntil, "Contract is locked");
24         uint256 amount = address(this).balance;
25         (bool success, ) = msg.sender.call{value: amount}("");
26         require(success, "Transfer failed");
27     }
28 }
```

Figure 5. Example of inconsistent access control vulnerability.

2.2. Current Smart Contract Vulnerability Detection Methods

Existing smart contract vulnerability detection methods can be roughly divided into two types: those based on traditional technologies, such as symbolic execution, fuzz testing, and formal verification, and vulnerability detection schemes based on deep learning technology.

Researchers have proposed a variety of vulnerability detection tools based on traditional methods to date. For example, Manticore is based on symbolic execution and taint analysis techniques, which accept contract source code or bytecode as input to detect integer overflow vulnerabilities [22]. The Oyente tool is also based on symbolic execution technology. It can detect vulnerabilities such as transaction sequence dependency, timestamp dependency, and re-entrancy. However, it has a high false-positive rate for integer overflow vulnerabilities [23]. Mythril, which is also based on symbolic execution technology, performs grammatical analysis on the smart contract code to determine the code structure and semantics, thereby building an abstract model. This model simulates the running state of the contract and performs checks on the model to identify possible security issues [24]. ContractFuzzer is based on fuzz testing technology, which can quickly generate a large amount of random data to test smart contracts to detect contract vulnerabilities quickly and effectively [25]. Solhint is based on formal verification technology. By performing lexical analysis and syntax analysis on the contract code and using predefined vulnerability rules and checkpoints to match the code, it can help developers identify code problems in the early stage of development [26]. Generally speaking, vulnerability detection technology based on traditional methods has the following bottlenecks:

(1) The degree of automation is low. Traditional methods must rely on expert experience to perform complex modeling of existing vulnerabilities and match them during vulnerability detection. For unmodeled vulnerabilities, the detection accuracy of traditional methods is unreliable. After a traditional method is tested, it is generally necessary to perform a manual audit.
(2) The accuracy rate is not high. When performing vulnerability detection through the superposition of hard rules, complex contracts may generate high false-positive rates, resulting in a decrease in accuracy.
(3) The detection time is extended. Most traditional methods are based on symbolic execution. When the code length is long, the number of execution paths increases exponentially, and the corresponding detection time is also lengthened, making it even more difficult to solve.

Some researchers introduced deep learning technology to propose new contract vulnerability detection schemes in response to the shortcomings of the abovementioned traditional methods. The authors of [27] considered the contract bytecode as input and proposed extraction of the connection between words based on the self-attention mechanism, extracting code features through CNN, and binary classification detection based three types of vulnerabilities: re-entrancy vulnerability, arithmetic vulnerability, and timestamp dependency. This method has a lower false-negative rate and average detection time than traditional methods. The authors of [28] proposed modeling vulnerabilities from source code based on the BiLSTM model. Although deep learning can automatically extract features from massive data and reduce manual analysis, the quality of features is related to the effect of the entire model. Therefore, some scholars have proposed different methods to enhance the feature extraction ability of the model. The authors of [29] proposed a novel AFS method from the perspective of input data to integrate various forms of feature information of the code; construct an abstract syntax tree by syntactically analyzing the contract; construct abstract syntax tree features through depth-first traversal; and use program slicing technology to program slicing, word segmentation vectorization, and syntax tree features that are spliced to increase the amount of information contained in the input features. The authors of [30] proposed a serial–parallel convolution bidirectional gated recursive network model from the perspective of network structure and extracted the characteristics of multivariable combinations through the serial–parallel convolution structure while retaining the call relationship and position relationship between contracts. The model proposed in [31] also

explores the best combination of different word embedding models and deep learning models from the perspective of model structure to obtain excellent contract code vector representations and improve classification accuracy. The authors of [32] proposed a method based on the BiLSTM model and a self-attention mechanism to extract critical features in the contract code. This method takes bytecode as input and does not analyze it from the source code perspective, nor does it consider whether the code feature vector can normally represent a certain category. At present, there is no way to optimize the model from the perspective of vector space representation, so we introduce a metric learning triplet loss function based on [32] so that contracts of the same category are aggregated in the vector space and contracts of different categories are discrete in the vector space. Vulnerability detection is carried out from the perspective of the source code, which is expected to help developers better detect vulnerabilities during code development.

3. Method

The solution proposed in this paper mainly involves analyzing the vulnerabilities at the source-code level and optimizing the feature representation ability of the BiLSTM model based on the triplet loss function and cross-entropy loss function to clearly distinguish the correct contract code from contracts with loopholes in the feature vector representation space. The model proposed in this paper is divided into five parts, as shown in Figure 6.

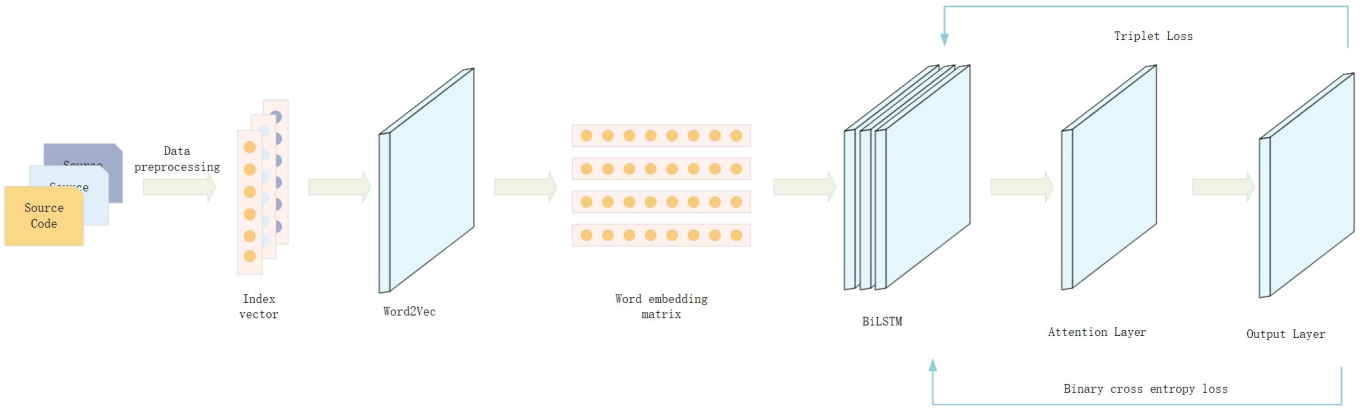

Figure 6. The overall architecture of the model.

- Data preprocessing: Perform data cleaning on the contract code, including removing irrelevant information, such as comments, versions, and variable names, to retain the core part of the code and perform word segmentation on it.
- Word-embedding layer: The token list obtained after word segmentation is converted into a word vector through the word-embedding model to represent the semantic information of the contract.
- BiLSTM layer: The feature representation of the contract code is extracted through the BiLSTM layer.
- Attention layer: To highlight the critical features in the contract code, we introduce an attention mechanism, using different weights to allocate the degree of attention to different features to improve the contract code classification performance.
- Vulnerability detection layer: Based on the binary classification loss function, a triplet loss function is introduced to optimize the feature representation ability through back-propagation. During the training process, by continuously updating the parameters, the normal contract code and the contract code with vulnerabilities can be better distinguished in the feature vector space, thereby improving the model's classification performance.

3.1. Data Preprocessing

A lot of information not related to vulnerabilities, such as comments, is contained in the source code of the smart contract we collected. The existence of such information adds noise to the vectorized representation of smart contracts and affect the classification effect. Therefore, we first clean the contract code according to the following steps (Figure 7).

1. Remove comments: Comments have nothing to do with code functions, so they can be removed from the code through regular expressions.
2. Remove useless characters such as spaces, tabs, and new lines: spaces, tabs, and new lines have no substantial impact on the semantics of the code, but they increase the dimensionality of the vectorized representation.
3. Remove the code compiler version information. Contracts usually specify the compiler version on the first line, which is not associated with a vulnerability.
4. Standardized code style: There may be different code styles in the smart contract source code, such as indentation, naming conventions, etc. To ensure the consistency of vectorized representation, we uniformly standardize the variable names or function names customized by developers as VAR plus numbers or FUN plus numbers.

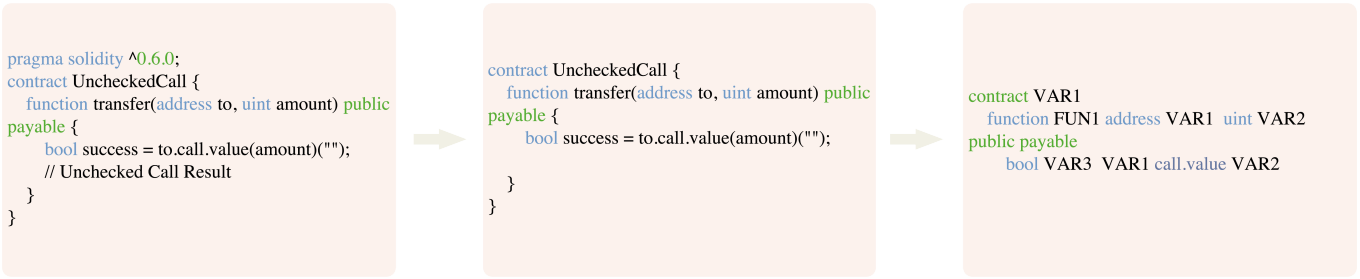

Figure 7. Source code processing.

3.2. Word-Embedding Layer

The input layer of the deep learning model usually requires the input to be a numerical vector, so it cannot directly accept the contract code as input data. The contract code needs to vectorize the text data first, divide the code into a list of words, then use the word-embedding model to map each word into a fixed-length vector as the model input. The specific processing flow is as follows:

1. Split code: We use regular expressions and spaces as separators to split the contract source code (C) into a word list: $T = [t1, t2, \ldots, tn]$;
2. Construct vocabulary: Construct a vocabulary based on the obtained word list, including all unique words that appear in the training data;
3. Word embedding: The word-embedding model can capture the semantic relationship between words and convert each word (t) into a fixed-length vector representation: $D = [v1, v2, \ldots, vn]$. The Word2Vec model is currently one of the most widely used word-embedding models, including two algorithms: Skip-Gram and Continuous Bag of Words (CBOW). Skip-Gram takes a word as input and predicts the context within a certain window. CBOW accepts the context within a certain window to predict the central word. Since CBOW uses the average value of the context, it converges faster than Skip-Gram. Considering that when the function of the contract is complex, the size of the code text increases, making the features more complex, the model proposed in this paper adopts the Word2Vec word-embedding model based on the CBOW algorithm;
4. Combined input: The word vector (D) is combined to form a feature representation of the contract code, usually using a convolutional or recurrent neural network structure to capture local or global features.

3.3. BiLSTM Layer

Based on the code vectorization representation obtained above, we use the BiLSTM model to effectively extract the semantic correlation features between contract codes and fully use the advantages of BiLSTM in capturing sequence relationships to better reveal the interrelationships between contract codes.

The BiLSTM model consists of two LSTM submodels, which capture forward and reverse semantic connections, respectively, as shown in Figure 8. The LSTM model introduces input gates, output gates, forget gates, and memory units on the basis of traditional recurrent neural network architectures to solve the gradient disappearance and gradient explosion problems in recurrent neural networks [33]. A schematic diagram of the LSTM is presented below (Figure 9).

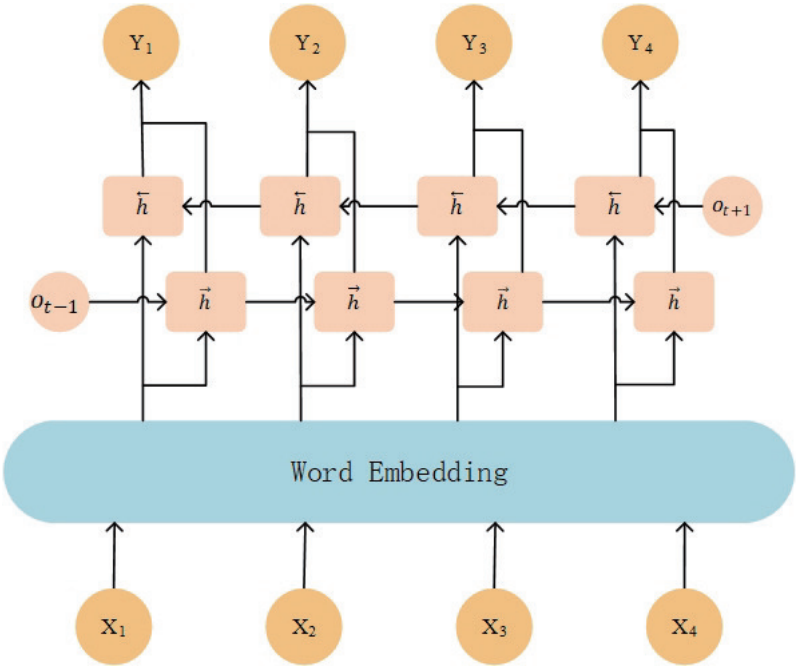

Figure 8. Diagram of the BiLSTM architecture.

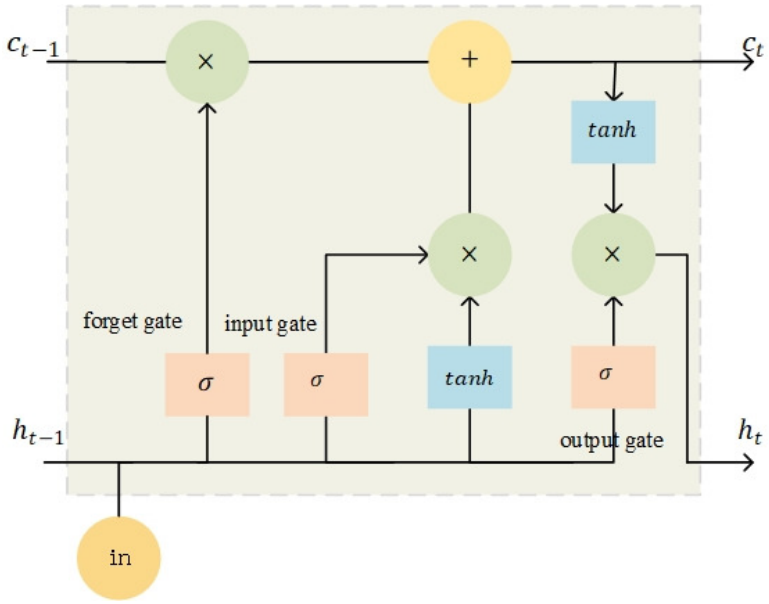

Figure 9. Schematic diagram of the LSTM network.

The forget gate determines which information is forgotten from the memory unit through the sigmoid function.

$$\mathrm{f_t} = \sigma(\mathrm{w_f} \cdot [h_{\mathrm{t}-1}, \mathrm{x_t}] + \mathrm{b_f}) \tag{1}$$

The input gate determines which information is added to the memory cell.

$$i_t = \sigma(w_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$

New candidate memory information is generated based on the previous hidden state (h_{t-1}) and the current input (x_t).

$$\tilde{c}_t = \tanh(w_c \cdot [h_{t-1}, x_t] + b_c) \tag{3}$$

The forget gate, previous memory cell, input gate, and candidate value are combined to update the current memory cell.

$$c_t = f_t * C_{t-1} + i_t * \tilde{c}_t \tag{4}$$

The output gate selects the input information from the current memory cell.

$$o_t = \sigma(w_o \cdot [h_{t-1}, x_t] + b_o) \tag{5}$$

The hidden state is updated according to the states of the output gates and memory cells.

$$h_t = o_t * \tanh(c_t) \tag{6}$$

BiLSTM stitches together the output of the hidden states through the forward and backward LSTM networks as the feature vector extracted from the BiLSTM source code, which effectively improves the detection accuracy of the model.

3.4. Attention Layer

BiLSTM has achieved significant improvements in solving the vanishing and exploding gradient problems of RNNs when processing long sequences but is still limited in capturing long-range dependencies. To solve this problem, we introduce the attention mechanism. The attention mechanism can also assign weights to each element in the sequence so that the model can adaptively focus on key parts during training, thereby improving performance.

In the attention layer, we first use the output features of the BiLSTM layer as input. Then, query, key, and value matrices are built for the input features. Next, the attention weights are calculated using the softmax function. Finally, the attention weights are multiplied by the value matrix to obtain an output vector that highlights key features. The following is the calculation formula for the attention layer:

$$\mathrm{Q} = \mathrm{X} * \mathrm{w_q} \tag{7}$$

$$\mathrm{K} = \mathrm{X} * \mathrm{W_k} \tag{8}$$

$$\mathrm{V} = \mathrm{X} * \mathrm{W_v} \tag{9}$$

$$\text{attention}_{\text{weights}} = \text{softmax}\big(\mathrm{Q} * \mathrm{K}\text{^}\mathrm{T}, \text{dim} = -1\big) \tag{10}$$

$$\text{attention}_{\text{output}} = \text{attention}_{\text{weights}} * V \tag{11}$$

3.5. Vulnerability Detection Layer Optimized by Triplet Loss

The vulnerability detection layer consists of a fully connected layer and an output layer. The output vector of the attention layer highlighting key features is used as input, and the digital label is used as the result of contract vulnerability detection. In this paper, we present binary classification model; 0 means that the contract has no loopholes, 1 means that the contract has loopholes, and the loss function usually uses the binary cross-entropy loss function.

$$\text{loss}_c(y, p) = -[y * \log(p) + (1 - y) * \log(1 - p)] \tag{12}$$

where y represents the true label of the sample, and p is the model-predicted label.

The model proposed in this paper adds a triplet loss function on the basis of the binary cross-entropy loss function. When classifying contracts, it improves the representation ability of the model, making the distance of the same class closer and the distance of different classes farther.

Triplet loss is a loss function for metric learning. During the training process, a map is constructed that can gather similar samples together in the feature space and separate dissimilar samples from each other. This loss function has achieved success in tasks such as computer vision and natural language processing, such as face recognition, recommender systems, and text similarity calculations [34].

The core idea of triplet loss is to construct "triplets". Each triplet contains an anchor sample, a positive sample, and a negative sample. Anchor points and positive samples belong to the same class, while anchor points and negative samples belong to different classes. The goal is to ensure that the distance between the anchor point and the positive sample is smaller than the distance between the anchor point and the negative sample, that is, it is hoped that in the feature space, the distance between similar samples is smaller than the distance between dissimilar samples. To achieve this goal, triplet loss defines an edge parameter (α), which indicates the minimum distance between the two. Therefore, the formula for triplet loss is:

$$\text{loss}_t(A, P, N) = \max\Big(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\Big) \tag{13}$$

where A, P, and N denote anchor points, positive samples, and negative samples, respectively, and $f(\cdot)$ denotes a function that maps input samples to feature space. The loss function requires that the distance from the anchor point to the positive piece plus the edge (α) be less than the distance from the anchor point to the negative selection. If this condition is not met, the value of the loss function is greater than 0. During the training process, the model attempts minimize this loss value to learn a suitable feature representation.

As shown in Figure 10, based on the different cases of the distance between the anchor point and the positive samples versus the distance between the anchor point and the negative samples, different strategies can be adopted for the construction of triads. The first strategy is to find the triplets that can be correctly classified in the training set, i.e., $\|f(A) - f(P)\|^2 < \|f(A) - f(N)\|^2$, called easy triplets. This strategy can lead to a model that easily learns a decision boundary, making the model overfit, reducing the generalization ability of the model, and decreasing the classification ability on new datasets. The second type of strategy involves finding triplets in the training set that are somewhat difficult but not too difficult, i.e., $\|f(A) - f(P)\|^2 < \|f(A) - f(N)\|^2 < \|f(A) - f(P)\|^2 + \alpha$. Semihard triplet mining can maintain better training stability while still focusing on the more difficult-to-distinguish samples. The third strategy is to find triplets in the training set that are difficult to classify correctly, i.e., $\|f(A) - f(P)\|^2 > \|f(A) - f(N)\|^2$, called hard triplet mining. This construction strategy allows the model to focus on learning the harder-to-distinguish samples and speeds up the convergence process but may also lead to training instability because the selected triads may be too difficult to learn. Another approach is random triplet mining, whereby triplets are randomly selected from the training set. This approach has the advantage of being simple to implement and stable in training but may lead to

slower convergence because the model spends more time learning samples that are easy to distinguish.

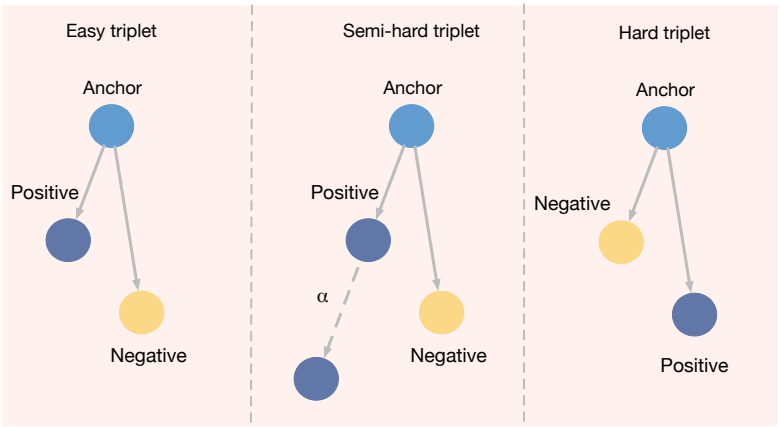

Figure 10. Triplet construction strategies.

According to the timing of triplet construction, it can be divided into offline triplet mining and online triplet mining. Offline triple mining refers to the construction triplets by selecting samples from the entire training set before the training process. Online triplet mining means that the model picks triplets during each iteration of training based on the current parameter state. This approach enables efficient training by adaptively selecting appropriate triplets according to the real-time performance of the model. In this study, we employ a strategy combining online triplet mining and semi-hard triplet mining to pick the best triplets during training. This approach enables the model to cluster similar data together while efficiently separating dissimilar data.

This study combines the binary cross-entropy loss function and the triplet loss function as the output layer's design to optimize the model's performance. The output layer takes the weighted sum of the two as the sum of the loss function of the model; the formula is as follows:

$$\text{loss} = \gamma_1 \, \text{loss}_c + \gamma_2 \, \text{loss}_t \tag{14}$$

4. Experiments

To verify the correctness and effectiveness of our proposed method, we conducted two sets of experiments. Experiment 1 comprised ablation experiments on the proposed model to verify the effect of the attention layer and triplet loss function. In Experiment 2, we compared the performance of the model proposed in this study with that of other models based on deep learning. The model proposed in this paper is based on the TensorFlow deep learning framework. The experimental operating environment was Windows 10 operated on an Intel Core i7-7700HQ CPU with 16GB RAM, and an NVIDIA GTX1050.

4.1. Dataset

We collected a large number of verified smart contracts from the Etherscan.io website, including contract addresses, bytecodes, and source codes. To further verify the accuracy of the contracts, we used the Slither, Smartcheck, and Oyente tools to classify the contracts and excluded contracts with inconsistent detection results. Finally, we selected four types of vulnerable contracts, including arithmetic vulnerabilities, re-entrancy vulnerabilities, unchecked calls, and inconsistent access control, as well as the security contract construction model dataset. The dataset contains 165,000 smart contracts, which we divided into a training set and a test set in a ratio of 8:2. The number of contracts contained in each vulnerability type is shown in Table 1.

Table 1. Statistics on the quantity of each type of contract.

	Arithmetic	Re-Entrancy	Unchecked Calls	Inconsistent Access Control	Security Contract
Amount	20,044	39,098	42,573	28,171	35,130

4.2. Evaluation Indicators

We used the four metrics of accuracy, precision, recall, and F1 score to evaluate the model's performance.

The accuracy rate indicates the ratio of the number of samples that the model predicts correctly to the total number of samples.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{15}$$

Precision represents the proportion of samples predicted to be positive relative to the actual proportion of positive samples. A high accuracy rate indicates that there are fewer false-positive samples and that the classifier has a strong ability to classify positive samples.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{16}$$

Recall represents the proportion of samples predicted to be positive among the samples that are actually positive. A high recall rate indicates that there are few false-negative samples and that the classifier has a strong coverage ability for positive samples.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{17}$$

The F1 score is the harmonic mean of precision and recall and is used to measure the performance of a model on an imbalanced dataset. A high F1 score indicates that the classifier has a strong ability to classify both positive and negative samples.

$$\text{F1 score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}} \tag{18}$$

TP refers to the number of positive samples predicted as positive samples, TN refers to the number of positive samples predicted as negative samples, FP refers to the number of negative samples predicted as positive samples, and FN refers to the number of negative samples predicated as positive samples.

4.3. Experimental Results

4.3.1. Ablation Experiment

Most of the vulnerabilities in smart contracts are gathered in one function, which is relatively concentrated. A large part of the features of the contract has nothing to do with the vulnerability, so the smart contract vulnerability detection scheme proposed in this paper adds an attention layer on the basis of BiLSTM, extracting the entire contract features. The attention layer assigns higher weights to vulnerability-related features to highlight important features. To improve classification accuracy, we also introduce triplet loss, which makes the model more concentrated in representing similar categories in the feature space while making different categories more dispersed in the feature space. To prove their role, we compared ANN, ANFIS, LSTM, BiLSTM, BiLSTM-ATT, and our scheme in detecting re-entrancy and arithmetic vulnerabilities. The experimental results are shown in Tables 2 and 3.

Table 2. Experimental results of BiLSTM, BiLSTM-ATT, and our model in detecting arithmetic vulnerability.

Model	Accuracy	Precision	Recall	F1 Score
ANN	55.43%	51.84%	49.02%	50.39%
ANFIS	58.73%	53.06%	51.93%	52.49%
LSTM	70.46%	65.31%	69.42%	67.30%
BiLSTM	79.94%	78.50%	77.03%	77.47%
BiLSTM-ATT	81.40%	81.53%	78.64%	80.93%
Ours	88.31%	86.34%	84.60%	85.46%

Table 3. Experimental results of BiLSTM, BiLSTM-ATT, and our model in detecting re-entrancy vulnerability.

Model	Accuracy	Precision	Recall	F1 Score
ANN	54.82%	51.61%	51.93%	51.77%
ANFIS	61.91%	57.37%	51.94%	54.52%
LSTM	72.31%	71.58%	69.93%	70.75%
BiLSTM	81.07%	78.93%	80.06%	79.71%
BiLSTM-ATT	88.21%	86.49%	84.21%	86.20%
Ours	93.25%	96.20%	86.13%	90.89%

As shown in Tables 2 and 3, the performance of the ANN and ANFIS models is relatively poor, with accuracy rates of only 55.43% and 58.73%, respectively, on the arithmetic vulnerability dataset. By adding contextual features, the accuracy and precision of the LSTM model are slightly improved. The BiLSTM model with the attention layer is more accurate than the single BiLSTM model. On the re-entrancy vulnerability dataset, the accuracy of the BiLSTM-ATT model is 7.16% higher than that of the single BiLSTM model, the accuracy is 7.56% higher, and the F1 score is 6.49% higher. On the arithmetic vulnerability dataset, although it is difficult to learn the characteristics of integer overflow and integer underflow vulnerabilities from the data, after adding the attention mechanism, the detection accuracy and precision of arithmetic vulnerabilities are improved.

Taking the re-entrancy vulnerability sample code as an example, the attention layer of the BiLSTM-ATT model assigns different weights to the contract code in vulnerability detection, and we generate a corresponding heat map based on the weights generated by it. As shown in Figure 11, the safe row is lighter in color, indicating that this example is less likely to have a vulnerability. The color of the re-entrancy line is darker, indicating that the weight assigned by the attention layer is greater in detecting re-entrancy vulnerabilities than in detecting arithmetic vulnerabilities, unchecked calls, and inconsistent access control vulnerabilities. The model considers this example to be at higher risk of re-entrancy vulnerabilities. In addition, the re-entrancy line gradually darkens from FUN2, which means that the weight assigned by the attention layer from FUN2 gradually increases. This shows that there is a greater possibility of re-entrancy vulnerabilities in this function, while the possibility of re-entrancy vulnerabilities in other functions is less. In addition, unchecked calls are similar to re-entrancy vulnerabilities and may be caused by low-level call functions, so the color in the heat map is darker than the that of the other two vulnerability categories. Therefore, the BiLSTM-ATT model introduces an attention layer and assigns different weights in vulnerability detection, which improves the detection accuracy and precision and enhances the visualization ability of the model.

Comparing our model with the single BiLSTM model, we can see that on the re-entrancy vulnerability dataset, our model has 12.18% higher accuracy, 17.27% higher precision, and a 12.13% higher F1 score than the single BiLSTM model. Compared with the BiLSTM-ATT model, we achieved 5.04% higher accuracy, 9.71% higher precision, and a 5.64% higher F1 score. Even on the arithmetic vulnerability dataset, our model

outperforms the single model and the BiLSTM-ATT model, with 6.91% higher accuracy, 4.81% higher precision, and 4% higher F1 score than the BiLSTM-ATT model.

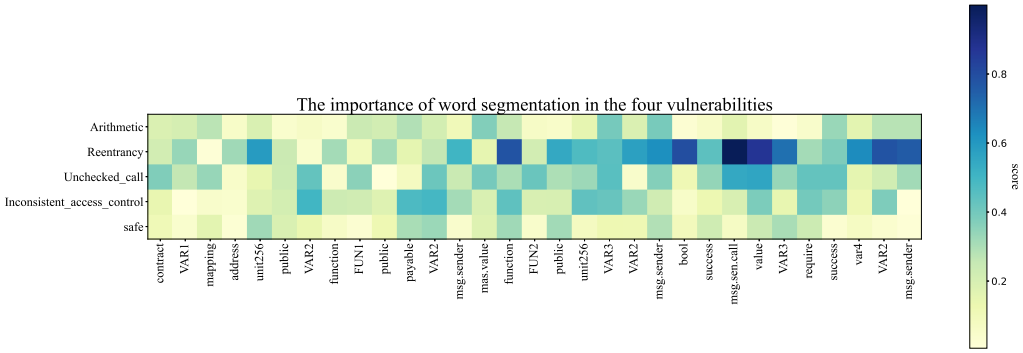

Figure 11. Feature weight heat map.

The above two figures show the visualization results of the BiLSTM-ATT model and our proposed model in three-dimensional space after performing t-SNE dimensionality reduction on the feature representation of contract codes containing arithmetic vulnerabilities. Figure 12a shows the feature extraction results of the BiLSTM-ATT model. There is an overlap between the contract code containing arithmetic vulnerabilities and the security code, and the classification effect needs to be improved. Figure 12b shows the feature extraction results of our proposed model after adding triplet loss. Compared with the previous model, the spatial representation of the two categories is more separated, and the classification effect is better. Therefore, triplet loss can make the features extracted by the model more cohesive in the same category of data and more dispersed in different categories of data, effectively improving the classification effect of the model.

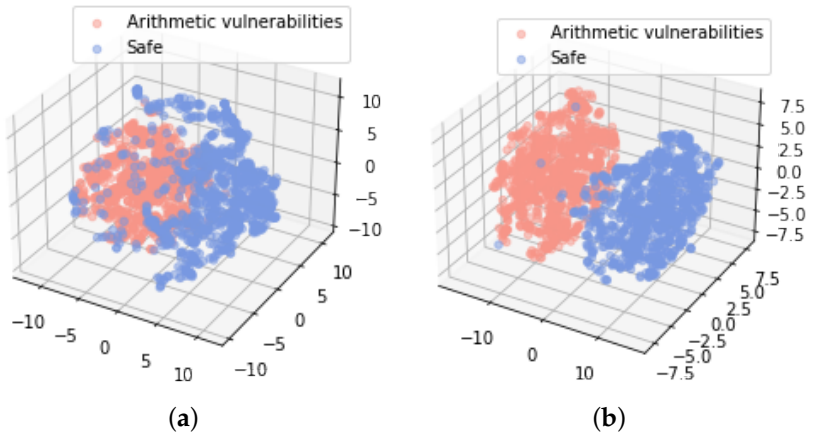

Figure 12. Feature vector visualization: (**a**) BiLSTM-ATT; (**b**) our model.

4.3.2. Comparative Experiment

To verify the detection performance of our proposed scheme in terms of smart contract vulnerabilities, we first compared it with static analysis tools such as Mythril and Oyente based on symbolic execution methods, which have been widely used. Tables 4 and 5 show the experimental comparison results of Mythril, Oyente, and our model on four kinds of vulnerabilities: arithmetic vulnerability, re-entrancy vulnerability, unchecked call vulnerability, and inconsistent access control.

Table 4. Comparison results between our model and traditional methods on arithmetic and re-entrancy datasets.

Model	Arithmetic				Re-Entrancy			
	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score
Mythril	61.53%	59.65%	52.63%	55.92%	60.01%	49.58%	51.69%	50.61%
Oyente	64.02%	61.35%	54.07%	57.48%	67.01%	53.52%	57.43%	55.41%
Ours	88.31%	86.34%	84.60%	85.46%	93.25%	96.20%	86.13%	90.89%

Table 5. Comparison results between our model and traditional methods on unchecked calls and inconsistent access control datasets.

Model	Unchecked Calls				Inconsistent Access Control			
	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score
Mythril	59.85%	52.04%	56.93%	54.38%	60.31%	54.91%	56.74%	55.81%
Oyente	68.01%	54.83%	61.04%	57.77%	63.92%	57.47%	57.06%	57.26%
Ours	91.85%	94.92%	90.06%	92.43%	90.59%	95.71%	86.13%	90.67%

As shown in Table 4, the model proposed in this paper has much higher accuracy and precision than the method based on symbolic execution. Most of the contracts used in the experiment have four to five functions. The contract code length is long, and the number is large. When the symbolic execution-based vulnerability detection tool performs symbolic analysis, the control flow path of the program increases dramatically. Therefore, in order to balance the execution efficiency and accuracy of the tools, such tools generally perform pruning to shorten the execution path, resulting in a considerably lower accuracy rate. In addition, the proposed model maintains around 90% accuracy in detecting re-entrancy vulnerabilities, unchecked calls, and inconsistent access controls and 88% accuracy in detecting arithmetic vulnerabilities. Because arithmetic vulnerabilities are generally caused by developers' ignorance of data types and operations, their performance is consistent with normal contracts, and it is difficult to distinguish them, so the correct rate is lower than that for other vulnerabilities. However, re-entrancy vulnerabilities and unchecked calls mostly occur when calling low-level functions such as calls, etc., which are easier-to-extract features and have a slightly higher detection accuracy. In addition, in terms of detection time, our model is much faster than tools based on symbolic execution. As shown in Table 6, traditional detection tools have an average detection time of around 5 seconds per contract, while our model maintains an average detection time per contract in the millisecond range, significantly improving detection speed.

Table 6. Comparison of detection time between our model and traditional models.

Average Time	Mythril	Oyente	Ours
Arithmetic	4.13 s	5.01 s	0.34 s
Re-entrancy	4.53 s	4.41 s	0.09 s
Unchecked calls	4.69 s	4.68 s	0.13 s
Inconsistent access control	4.57 s	5.01 s	0.12 s

In addition, we compared the performance of our model with that of other deep learning models, including TextCNN, RNN, LSTM-ATT, GRU, CodeBERT, and XLNet models. The experimental results are shown in Figures 13–16. TextCNN performs slightly better than the other four models on the four vulnerability datasets, followed by the LSTM-ATT model, GRU model, LSTM model, and RNN model. Using arithmetic vulnerabilities as an example, the accuracy and precision of the RNN model are both low, and the accuracy is the same as that of the Mythril tool based on symbolic execution (only 61.58%),

and the precisionis only 61.03%. Because most of the contract datasets we collected are long contract codes, the dataset comprises a large number of contracts. When the training contract code size is large, the RNN model gradient disappears or explodes, and the accuracy is affected and greatly reduced. The LSTM model improves the accuracy rate of arithmetic loopholes (70.46%) by introducing the memory unit mechanism and the gating unit, the precision rate is 65.31%, the recall rate is 69.24%, and the F1-score is 67.30%. The accuracy rate of the GRU model is 71.07%, the precision rate is 66.19%, the recall rate is 70.75%, and the F1-score is 68.39%. However, when the code length is longer, although the LSTM and GRU models are better than the RNN models, the accuracy and F1 score still decrease when they learn longer dependencies. The LSTM-ATT model with the attention mechanism can slightly improve the performance, with an accuracy rate of 73.41%, a precision rate of 72.51%, a recall rate of 69.67%, and an F1 score of 71.06%. Our model performs best on arithmetic loopholes, with an accuracy rate of 88.31%, a precision score of 86.34%, a recall rate of 84.60%, and an F1 score of 85.46%, effectively improving the classification effect of positive and negative classes. Compared with the most advanced code language models, the CodeBERT model and the XLNet model, the accuracy and precision of our model are 9.64% and 12.19% higher on the arithmetic vulnerability dataset and 6.83% and 7.13% higher on the re-entrancy vulnerability dataset, respectively.

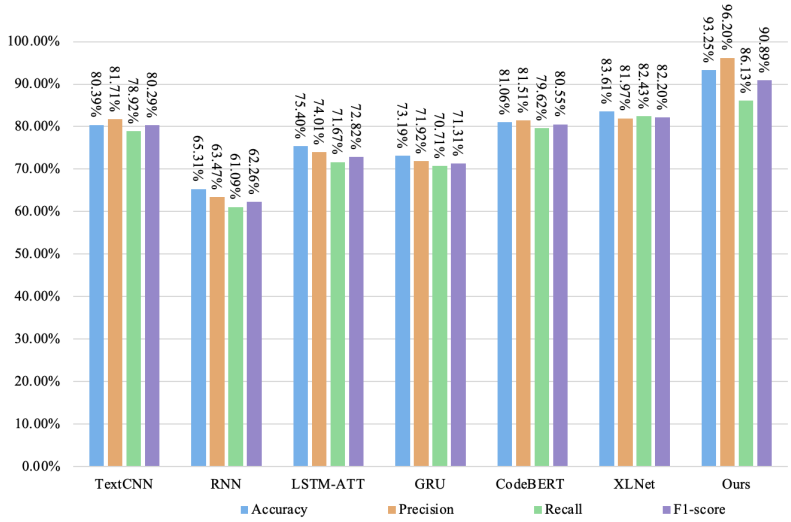

Figure 13. The detection results of each model on the arithmetic vulnerability dataset.

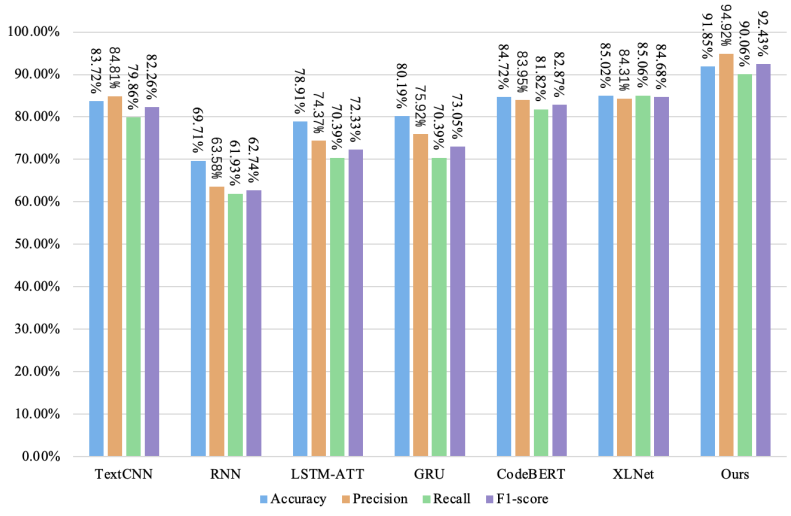

Figure 14. The detection results of each model on the re-entrancy vulnerability dataset.

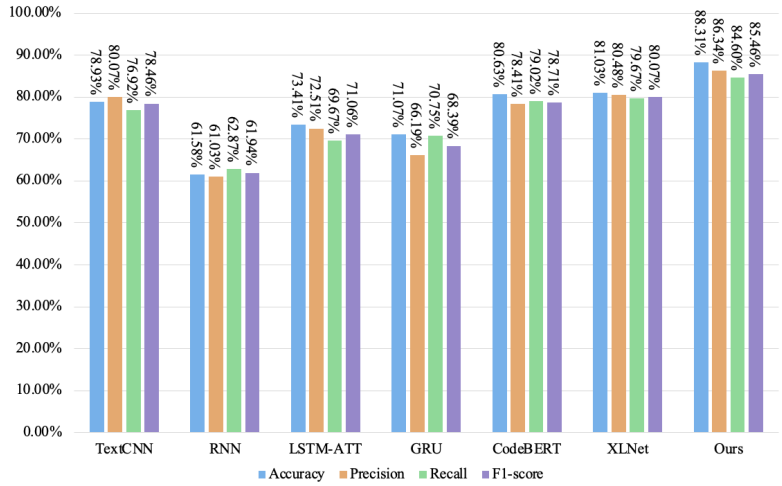

Figure 15. The detection results of each model on the unchecked call vulnerability dataset.

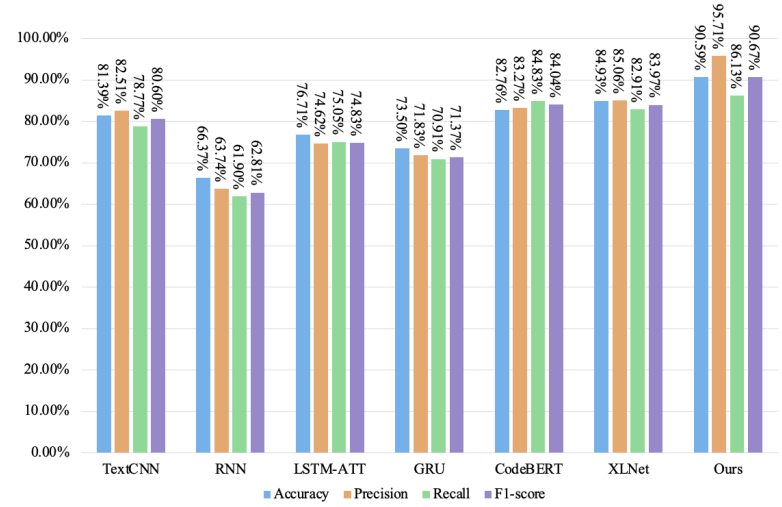

Figure 16. The detection results of each model on the inconsistent access control vulnerability dataset.

5. Conclusions

In this paper, we propose a scheme for smart contract vulnerability detection based on metric learning and the bidirectional long short-term memory network model.

(1) We propose a new vulnerability detection model from the perspective of feature representation space optimization. In contrast to existing methods, we do not improve the feature learning ability of the whole model by combining multiple models but introduce the metric learning triplet loss function on the basis of the traditional binary cross-entropy loss function to optimize the feature representation ability of the model. By optimizing the feature space, contracts of the same category are closer and contracts of different categories are further apart so as to improve the detection accuracy of the model and make up for the limitation of the high false-positive rate of the traditional method.

(2) The proposed scheme enhances the interpretability of vulnerability detection. We use the source code of the smart contract as the input data, and through the word vectorization and attention mechanism, we can screen out the key features of the vulnerability and locate the cause of the smart contract vulnerability in the source code.

(3) We constructed a large-scale dataset for contract source code vulnerability detection. We collected 165,000 verified contracts from the Etherscan website and used tools such as Slither and Mythril to classify the contracts.

(4) The scheme has excellent detection performance. To fairly evaluate the performance of our model, we compared it with traditional methods based on symbolic analysis and other deep-learning-based methods on four datasets of arithmetic vulnerabilities, reentrancy vulnerabilities, unchecked calls, and inconsistent access control. The results show that our model achieves the best accuracy on these four datasets (88.31%, 93.25%, 91.85%, and 90.59%, respectively), while maintaining a high F1 score of 85.46%, 90.89%, 92.43%, and 90.67%, respectively. Compared with symbolic analysis methods, our model has a faster detection speed. According to the experimental results, our proposed model has better classification performance and detection speed.

However, our current model can only judge whether there is a vulnerability in the contract and cannot determine the specific type of vulnerability. In the future, we will continue to study how to simultaneously introduce metric learning and multilabel classification models in smart contract vulnerability detection to identify specific vulnerability categories.

Author Contributions: Conceptualization, M.W. and K.Z.; methodology, M.W.; validation, M.W., X.W., and K.Z.; writing—original draft preparation, M.W.; writing—review and editing, M.W.; visualization, M.W. and Z.X.; supervision, K.Z.; project administration, M.W. and J.L.; funding acquisition, K.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National key research and development plan based on "Internet +" village community public service enhancement technology research (grant number 2019YFD1101104).

Data Availability Statement: The experimental code and data used in this study will be posted on the dead simple repository (https://github.com/SunnyWang01/SmartContractDetect, (accessed on 10 January 2023)).

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

References

1. Wang, S.; Ouyang, L.; Yuan, Y.; Ni, X.; Han, X.; Wang, F.Y. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Trans. Syst. Man Cybern. Syst.* **2019**, *49*, 49–77. [CrossRef]
2. Capocasale, V.; Perboli, G. Standardizing smart contracts. *IEEE Access* **2022**, *10*, 91203–91212. [CrossRef]
3. Ivashchenko, N.P.; Shastitko, A.Y.; Shpakova, A.A. Smart contracts throught lens of the new institutional economics. *J. Institutional Stud.* **2019**, *11*, 64–83. [CrossRef]
4. Sharma, A.; Tomar, R.; Chilamkurti, N.; Kim, B.G. Blockchain based smart contracts for internet of medical things in e-healthcare. *Electronics* **2020**, *9*, 1609. [CrossRef]
5. Lu, J.; Wu, S.; Cheng, H.; Song, B.; Xiang, Z. Smart contract for electricity transactions and charge settlements using blockchain. *Appl. Stoch. Model. Bus. Ind.* **2021**, *37*, 37–53. [CrossRef]
6. Goudarzi, A.; Ghayoor, F.; Waseem, M.; Fahad, S.; Traore, I. A Survey on IoT-Enabled Smart Grids: Emerging, Applications, Challenges, and Outlook. *Energies* **2022**, *15*, 6984. [CrossRef]
7. Waseem, M.; Adnan Khan, M.; Goudarzi, A.; Fahad, S.; Sajjad, I.A.; Siano, P. Incorporation of Blockchain Technology for Different Smart Grid Applications: Architecture, Prospects, and Challenges. *Energies* **2023**, *16*, 820. [CrossRef]
8. Kumar, P.; Kumar, R.; Gupta, G.P.; Tripathi, R. A Distributed framework for detecting DDoS attacks in smart contract-based Blockchain-IoT Systems by leveraging Fog computing. *Trans. Emerg. Telecommun. Technol.* **2021**, *32*, e4112. [CrossRef]
9. Zhou, Q.; Zheng, K.; Zhang, K.; Hou, L.; Wang, X. Vulnerability Analysis of Smart Contract for Blockchain-Based IoT Applications: A Machine Learning Approach. *IEEE Internet Things J.* **2022**, *9*, 24695–24707. [CrossRef]
10. Gupta, R.; Patel, M.M.; Shukla, A.; Tanwar, S. Deep learning-based malicious smart contract detection scheme for internet of things environment. *Comput. Electr. Eng.* **2022**, *97*, 107583. [CrossRef]
11. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, W.; Chen, X.; Weng, J.; Imran, M. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* **2020**, *105*, 475–491. [CrossRef]
12. Ullah, F.;Al-Turjman, F. A conceptual framework for blockchain smart contract adoption to manage real estate deals in smart cities. *Neural Comput. Appl.* **2021**, *35*, 1–22. [CrossRef]
13. 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.* **2020**, *8*, 1133–1144. [CrossRef]
14. Wang, X.; Sun, J.; Hu, C.; Yu, P.; Zhang, B.; Hou, D. EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, 1565007. [CrossRef]
15. Sun, T.; Yu, W. A formal verification framework for security issues of blockchain smart contracts. *Electronics* **2020**, *9*, 255. [CrossRef]

16. Shafay, M.; Ahmad, R.W.; Salah, K.; Yaqoob, I.; Jayaraman, R.;Omar, M. Blockchain for deep learning: Review and open challenges. *Clust. Comput.* **2022**, *14*, 1–25.
17. Cai, J.; Li, B.; Zhang, J.; Sun, X.; Chen, B. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *J. Syst. Softw.* **2023**, *195*, 111550. [CrossRef]
18. Dai, M.; Yang, Z.; Guo, J. SuperDetector: A Framework for Performance Detection on Vulnerabilities of Smart Contracts. *J. Phys. Conf. Ser.* **2022**, *2289*, 012010. [CrossRef]
19. Zhang, L.; Wang, J.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. A novel smart contract vulnerability detection method based on information graph and ensemble learning. *Sensors* **2022**, *22*, 3581. [CrossRef]
20. Liu, Z.; Qian, P.; Wang, X.; Zhuang, Y.; Qiu, L.; Wang, X. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans. Knowl. Data Eng.* **2021**, *35*, 1296–1310. [CrossRef]
21. Ye, J.; Ma, M.; Lin, Y.; Ma, L.; Xue, Y.; Zhao, J. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *J. Syst. Softw.* **2022**, *192*, 111410. [CrossRef]
22. Mossberg, M.; Manzano, F.; Hennenfent, E.; Groce, A.; Grieco, G.; Feist, J.; Brunson, T.; Dinaburg, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1186–1189.
23. Perez, D.; Livshits, B. Smart contract vulnerabilities: Does anyone care? *arXiv* **2019**, arXiv:1902.06710.
24. Mueller, B. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amst.* **2018**, *9*, 54.
25. Jiang, B.; Liu, Y.; Chan, W.K. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Leipzig, Germany, 3–7 September 2018; pp. 259–269.
26. Abdellatif, T.; Brousmiche, K.L. Formal verification of smart contracts based on users and blockchain behaviors models. In Proceedings of the 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Lisbon, Portugal, 26–28 February 2018; pp. 1–5.
27. Sun, Y.; Gu, L. Attention-based machine learning model for smart contract vulnerability detection. *J. Phys. Conf. Ser.* **2021**, *1820*, 012004. [CrossRef]
28. Zhang, X.; Li, J.; Wang, X. Smart Contract Vulnerability Detection Method based on Bi-LSTM Neural Network. In Proceedings of the 2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA), Dalian, China, 20 August 2022; pp. 38–41.
29. Wang, B.; Chu, H.; Zhang, P.; Dong, H. Smart Contract Vulnerability Detection Using Code Representation Fusion. In Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taiwan, China, 6 December 2021; pp. 564–565.
30. Zhang, L.; Li, Y.; Jin, T.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. SPCBIG-EC: A robust serial hybrid model for smart contract vulnerability detection. *Sensors* **2022**, *22*, 4621. [CrossRef] [PubMed]
31. 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]
32. 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. [CrossRef]
33. Graves, A.; Graves, A. Long short-term memory. In *Supervised Sequence Labelling with Recurrent Neural Networks*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 37–45.
34. Hoffer, E.; Ailon, N. Deep metric learning using triplet network. In *Similarity-Based Pattern Recognition: Third International Workshop, SIMBAD 2015, Copenhagen, Denmark, 12–14 October 2015; Proceedings 3*; Springer International Publishing: Berlin/Heidelberg, Germany, 2015; pp. 84–92.