Journal of Computer Science and Artificial Intelligence
ISSN: 3078-8242 (Print), ISSN: 3078-2309 (Online) | Vol. 2, No. 2, 2025

Smart Contract Vulnerability Detection Method Based on Deep and Cross Network with Feature Aggregation

Peng Zhao[1], Jinsheng Li[1], Shuaijun Gao[1], Jie Zhao[1], Fang Liang[2,*]

[1]School of Computer Science and Technology, Taiyuan Normal University, Jinzhong, 030600, China
[2]Ningbo Vocational and Technical Education Center School, Ningbo City, 315000, China
[*] Corresponding author

Abstract: The advent of decentralised applications across a range of sectors has led to a growing emphasis on the research and development of methods to identify vulnerabilities in smart contracts for decentralised applications. However, current detection techniques have been found to have limitations in terms of accuracy and the number of false alarms they generate. In order to address the aforementioned issues, this paper puts forth a modular vulnerability detection model, designated as BAMC. The method initially utilises the word2vec model to derive the word vector representation of the smart contract, subsequently extracting the word order information through a bidirectional long short-term memory network. Subsequently, the attention mechanism and max-pooling operation are employed to process the word order information, thereby obtaining fine-grained features and key features. Ultimately, explicit bounded-degree feature interactions are achieved through the combination of deep and cross networks, thus enabling the detection of reentrancy vulnerabilities and timestamp vulnerabilities. The experimental results demonstrate that the proposed method exhibits superior performance in comparison to existing techniques, with significantly higher values for various indexes. Notably, the reentrancy vulnerability and the F_1-score of timestamp vulnerability reach 86.14% and 91.43%, respectively.

Keywords: Blockchain; Smart Contracts; Deep Learning; Vulnerability Detection; Deep and Cross Network.

1. Introduction

A smart contract is a computer protocol designed to communicate, verify, or execute contracts in an information-based manner. Smart contracts enable trusted transactions to be conducted without the intervention of a third party, and these transactions are traceable and irreversible. The concept was proposed by computer scientist Nick Szabo in 1994[1]. In recent years, with the continuous development of blockchain technology, smart contracts have also successfully demonstrated their unique advantages in fairness and reliability in application models such as equity management, financial products, games, and insurance.

Smart contracts are increasingly being used on blockchain platforms, and it is estimated that the virtual value covered by smart contracts has reached 10 billion US dollars. Since smart contracts usually involve the management of large amounts of funds and digital assets, they have become attractive targets for cyber attackers. Smart contracts are inherently irrevocable and unmodifiable, and once a vulnerability occurs, the losses caused cannot be suspended or rolled back. For example, in 2016, hackers stole tens of millions of dollars worth of Ether from the DAO investment fund through a reentrancy attack, which also led to the fork of the Ethereum blockchain[2]. In 2018, attackers launched an attack on the integer overflow vulnerability in the ERC20 protocol, causing the market value of the BEC coin to almost zero[3]. According to statistics from the SharkTeam team, the losses caused by smart contract vulnerabilities in 2022 alone reached 1.7 billion US dollars[4]. Therefore, it is particularly important to establish an efficient security detection solution. Solidity is the current mainstream smart contract programming language, and its widespread application makes it an urgent need to ensure code security. This study uses Solidity source code as the basic dataset and introduces deep learning technology to solve the problem that traditional methods are inefficient in detecting reentrancy vulnerabilities and timestamp vulnerabilities.

2. Related Work

Traditional technologies for smart contract vulnerability detection include fuzz testing, symbolic execution, formal verification, static analysis and other vulnerability detection methods. Nguyen et al. [5] proposed the sFuzz model, which is an efficient adaptive fuzz testing tool specifically for Solidity smart contracts. It combines the strategies of the AFL fuzz tester and the lightweight multi-target adaptive strategy to target code branches that are difficult to cover, thereby improving test efficiency and code quality. Luu et al. [6] proposed the Oyente method, which uses the contract control flow graph to construct symbolic states and uses the Z3 solver to obtain symbolic execution paths, thereby realizing the detection of smart contract security vulnerabilities. The Solidifier formal verification method [7] uses Corral to perform contract verification and function verification on the Boogie intermediate verification language converted from Solidity source code to ensure the security of the contract. Tikhomirov et al. [8] proposed an extensible static analysis tool SmartCheck, which uses XPath query statements to match vulnerabilities that conform to expert mode rules on the XML parse tree converted from source code.

These traditional methods are often based on fixed rules, and these methods themselves also face significant limitations. For example, fuzz testing has limited path coverage, making it difficult to fully explore all execution paths of a contract. Symbolic execution faces the problem of path explosion. As the complexity of the contract increases, too many execution paths may be generated, making symbolic execution impractical or insufficient computing resources. Although formal verification can provide high security guarantees, it has a low degree of automation and requires manual

intervention for specification and proof. Static analysis tools can identify potential problems at compile time, but they have incomplete information problems for contracts that dynamically generate code or load modules.

In recent years, deep learning has made great progress in natural language processing methods, and the emergence of ChatGPT has significantly promoted the development of AIGC. The application of deep learning techniques to the field of vulnerability detection has also achieved some results, Zhuang et al. [9] proposed to construct a degree-free graph neural network (DR-GCN) and a temporal message propagation network (TMP) for vulnerability detection from designed contract graphs.Li et al. [10] used graph features extracted from contracts, combined with features in expert mode, and performed feature interaction after dimensionality reduction to detect the existence of vulnerabilities. The ReChecker method [11] slices contracts and utilizes a bidirectional long short memory network and an attention mechanism to accurately detect reentrancy vulnerabilities. Wang et al. [12] built a variety of machine learning models for vulnerability detection by using n-gram technology to extract features from the bytecode of contracts. Although these methods are helpful in solving the problem of detection efficiency, most of them use intermediate representation rather than source code, which may lead to the loss of some syntax and semantic information during the conversion process. Furthermore, most adopt a single network approach, which runs the risk of learning insufficient critical information. In order to deal with the above challenges, this paper proposes a smart contract vulnerability detection method based on feature aggregation deep cross network, aiming to improve the accuracy of smart contract vulnerability detection.

3. Our Approach

This paper builds a smart contract vulnerability detection model based on the embedding layer, BiLSTM layer, max-pooling layer, attention layer, deep and cross network layer and vulnerability detection layer, realizing the explicit bounded feature interaction of context-aware fine-grained features and the most critical features.

First, the Solidity source code is converted into a vector representation that the machine is good at processing in the embedding layer. In the execution phase of the Solidity source code, the code is executed line by line in the order in which it is written. This sequential execution not only affects the execution result of each instruction, but also retains the context information of the previous instruction. The BiLSTM model is good at capturing the dependencies in this time series data. Inputting the vector output by the embedding layer into the BiLSTM layer can effectively extract the word order feature vector of the smart contract. Vulnerabilities in smart contracts often arise from logical errors in a few lines of code. Compared with the extensive review of the entire code, detailed analysis of specific logic fragments can often more effectively identify potential security vulnerabilities. On the other hand, vulnerabilities in smart contracts usually occur in the process of calling functions in multiple contract interactions. Therefore, the word order feature vectors obtained in the BiLSTM layer are input into the maximum pooling layer and the attention layer respectively. The maximum pooling layer focuses on the local key features that can extract the contract vulnerabilities. The attention mechanism shows significant effectiveness in capturing the complex relationship between each interaction in the smart contract, making it possible to extract global fine-grained features. The combination of these two methods allows us to analyze the security of the contract from different levels. Next, the deep cross network layer obtains the interaction information between each feature vector in the contract to provide richer data support for the analysis and decision-making of the smart contract. The key features and fine-grained features are spliced and passed to the deep cross network layer. Finally, the output results are passed through the fully connected layer in the vulnerability detection layer to obtain the final vulnerability detection results. The implementation process of the above layers will be described in detail below. The overall vulnerability detection framework is shown in Figure 1.

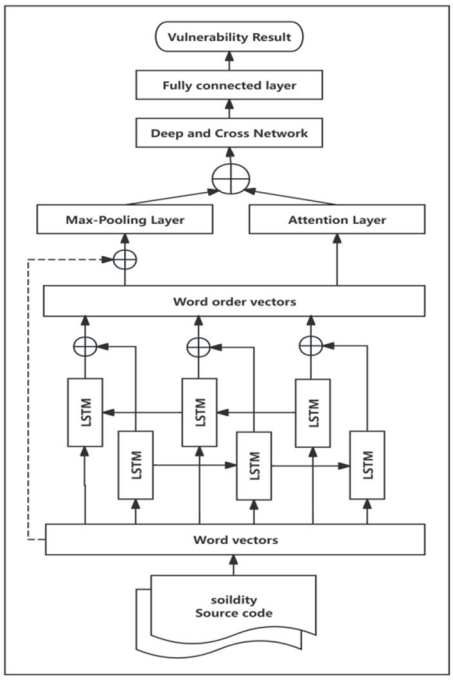

Figure 1. Smart contract vulnerability detection framework based on deep and cross network with feature aggregation.

3.1. Embedded layer

The embedding layer converts the Solidity source code into a low-dimensional dense vector of fixed length that can express semantic relationships. One-hot encoding is a simple and intuitive text representation method, but it has the problem of dimensionality explosion and inability to capture the semantic relationship of words. The Word2Vec model proposed by MIKOLOV et al. [13] can solve these problems. The Word2Vec model includes two modules: Skip-Gram and CBOW. The Skip-gram model predicts the context words around a word given a word, and the CBOW model predicts the center word given other words in the window. This paper uses the Skip-gram model to embed the source code. Figure 2 shows the spatial distribution after mapping the word vector to a two-dimensional space. It can be observed from Figure 2 that the word vectors of the same category show a relatively close spatial distance . For example, the positions of modifier types such as "payable", "require", and "external" and operation types such as ">", "<", and "&" are relatively close. This proves that the semantic relevance of word vectors is effectively preserved by this method.

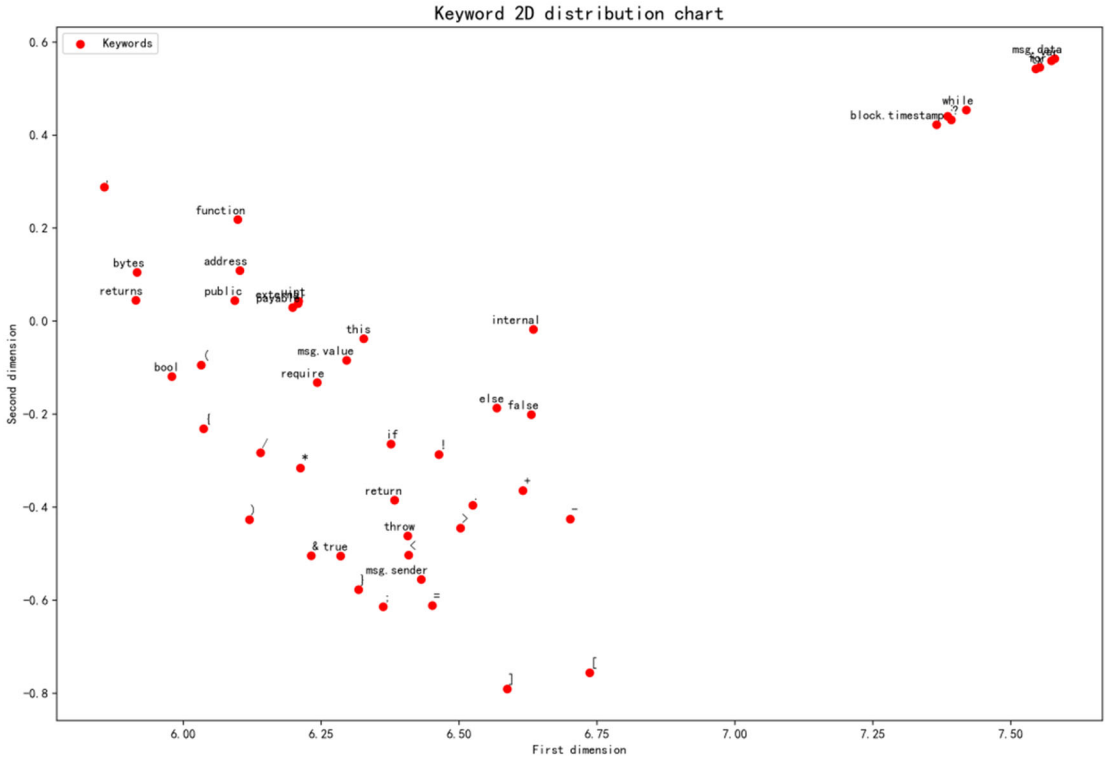

Figure 2. Keyword 2D distribution chart

3.2. BiLSTM layer

The BiLSTM model comprises two independent LSTMs[14], which model sequence information from the forward and backward perspectives, respectively. This approach allows for the more accurate capture of long-distance bidirectional dependencies. In comparison to traditional recurrent neural networks, LSTM incorporates a gating mechanism to address the issue of gradient explosion or gradient disappearance. In the cell unit c_t of an LSTM, key information can be stored and transmitted for an extended period of time, and key information can be updated in cell unit c_t at different time steps. The updated cell unit c_t information is then conveyed to the hidden state of the current time step and subsequently transmitted to the next time step unit. As illustrated in the subsequent formula:

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad (1)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad (2)$$

$$h_t = o_t \odot \tanh(c_t) \quad (3)$$

In the formula, the letters W, U, and b represent the parameters that must be trained in the network, while $\odot$ represents the bit-by-bit multiplication of the elements. $\tilde{c}_t$ represents the candidate state of the cell at time t, which is obtained by performing the $\tanh(\cdot)$ function operation on the smart contract code x_t at time t and the hidden state at time t-1. f_t, i_t, and o_t are the gating mechanisms of the network. The forget gate f_t determines which information from the cell unit at the previous moment is discarded. The input gate i_t determines which information is stored in the memory unit at the current moment. o_t is the output gate, which obtains the hidden state at the current moment by operating with the cell unit c_t. The following formulas represent the calculations for the input gate, forget gate, and output gate:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (4)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (5)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (6)$$

In the above formula, $\sigma(\cdot)$ is the sigmoid activation function, and x_t is the smart contract code input at time t. The sigmoid function controls the value between (0,1) to realize which information is forgotten at the previous moment and how much information should be retained in the current cell candidate state. In BiLSTM, the word vector is passed to the forward and backward LSTM respectively, and the corresponding hidden state is concatenated to obtain the BiLSTM vector.

3.3. Max-Pooling layer

The BiLSTM model, while endeavoring to overcome the limitation of unidirectional propagation of information when dealing with sequential data, still suffers from the bias problem of later words being more dominant than earlier ones. Since the vulnerability can occur anywhere in the code, the unbiased component of convolutional neural network is introduced to implement the maximum pooling operation[15]. By implementing the maximal stratification operation the discriminators in the smart contract can be selected fairly. The process of this operation is shown in Figure 3.

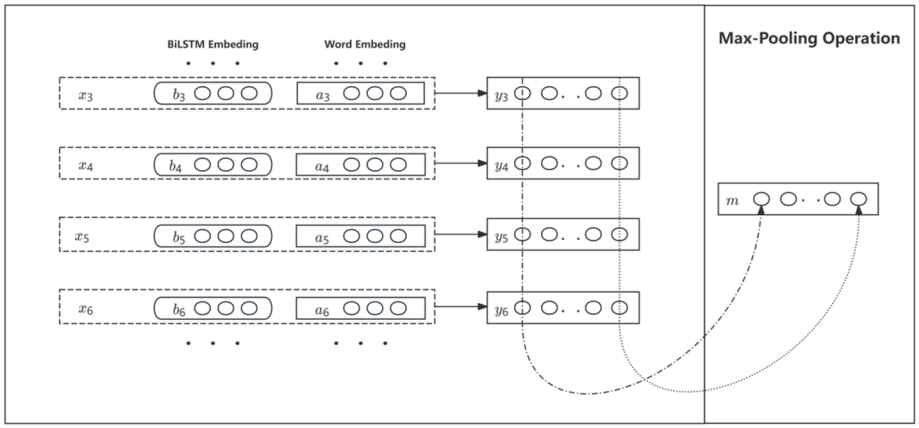

Figure 3. Structure of the max-pooling layer

The output b_i of the BiLSTM layer is concatenated with the output a_i of the word embedding layer to obtain x_i, where i represents the i-th word in the text, as illustrated in Formula (7). In this manner, the utilization of contextual information enables our model to more effectively resolve word ambiguity in comparison to traditional neural models that rely on partial text information.The resulting x_i is nonlinearly transformed by the $\tanh(\cdot)$ activation function, and then the processed result is subjected to the max-pooling operation, As shown in the Formula (8):

$$x_i = [b_i\,;\, a_i] \tag{7}$$

$$y_i = \tanh\left(W_y X_i + b_y\right) \tag{8}$$

$y_i$is a semantic vector space that is capable of capturing more complex data models and enhancing the model's expressiveness. The max-pooling operation is employed to extract the most salient features from y_i, As illustrated in Formula (9):

$$m = \max_{i=1}^{n}(y_i) \tag{9}$$

The $\max(\cdot)$ function is a selection function that returns the maximum value in the input, where n represents a fixed sequence length. The k -th value of m represents the maximum value of the k-th value in y_i. With regard to the selection of pooling layer types, in addition to max-pooling, other operations such as average pooling are available. However, this particular operation is not selected due to the potential for a few lines of code to introduce vulnerabilities. The max-pooling layer is capable of identifying the most critical potential factors.

3.4. Attention layer

The objective of the attention layer is to obtain detailed feature information. The maximum pooling layer is only capable of retrieving the maximum value within a given feature, which presents a challenge in integrating the interaction information between disparate functions within a smart contract. The majority of vulnerabilities associated with smart contracts pertain to intricate interconnections between multiple contract functions. From a comprehensive global information standpoint, the capture of global detailed features enables a more nuanced understanding of the interaction information between various functions within the code. Accordingly, this paper employs the attention mechanism[16] to facilitate the automatic acquisition of fine-grained information within the context of smart contracts. Initially, the output $B = [b_1, b_2, \ldots, b_n]$ of the BiLSTM layer is subjected to a non-linear operation in order to obtain the attention k vector, as illustrated in Formula (10). Subsequently, the attention feature vector v is derived by multiplying the attention weight, learned by the $\mathrm{softmax}(\cdot)$ function, with the input sequence element, B, bit by bit, As illustrated in Formula (11).

$$k = \tanh(W_k B) \tag{10}$$

$$v = softmax\left(W_q k\right) \odot B \tag{11}$$

3.5. Deep and cross network layer

The deep and cross network layer provides additional interactive information beyond that of single features[17].For example, the "call.value" keyword and the "−=" subtraction operation keyword in the contract are one of the common reentrancy vulnerabilities triggering conditions. The interactive information associated with these two elements is more informative than any single feature. The core structure of the deep and cross network is depicted in Figure 4. The input layer vector is derived by fusing the output of the max-pooling layer and the output of the attention layer, that is, $x_0 = [m\,;v]$.

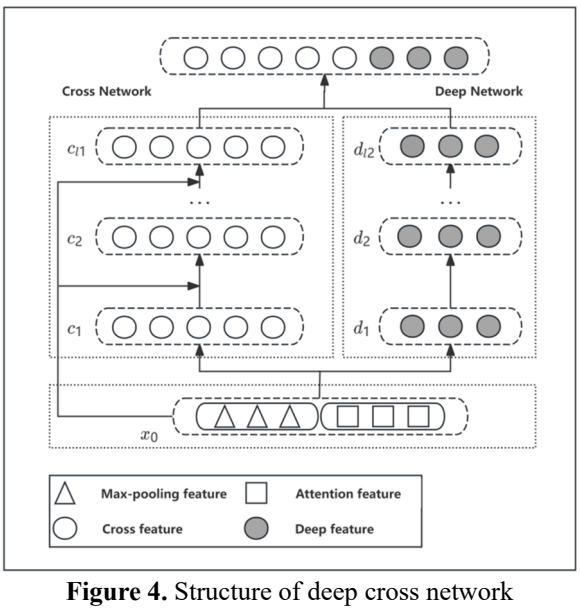

Figure 4. Structure of deep cross network

The cross neural network represents the core component of the deep cross network layer, facilitating the effective learning of feature combinations and enabling the generation of explicit high-order cross features. c_l and c_{l+1} represent the input and output of the $(l+1)$-th layer of the cross network, whereas W_l and b_l denote the learning weight matrix and parameter vector of the $(l+1)$-th layer. As illustrated in the subsequent formula:

$$c_{l+1} = x_0 \odot (W_l c_l + b_l) + c_l \tag{12}$$

As illustrated in Formula (12), the cross feature order of the cross neural network is inextricably linked to the number of layers in the designed cross layer. This implies that the network is constrained to polynomial functions within a specified range, yet there are inherent limitations on the modeling of other complex function spaces, necessitating approximate processing. Accordingly, to more accurately model the inherent distribution in the data, a deep neural network is introduced as a supplement to generalize its modeling ability. In this context, d_l and d_{l+1} represent the input and output of the $(l+1)$-th layer of the cross network, W_l and b_l denote the learning weight matrix and bias vector of the $(l+1)$-th layer, and $\mathrm{ReLU}(\cdot)$ denotes the ReLU activation function. The results of the cross layer and the deep layer are then concatenated to obtain the final output, as illustrated by the following formula:

$$d_{l+1} = ReLU(W_l d_l + b_l) \tag{13}$$

$$x_{final} = [c_{l1}; d_{l2}] \tag{14}$$

3.6. Vulnerability detection layer

In this layer, a fully connected layer is employed as the classifier. The output feature vector of the deep cross network is input into the fully connected layer for dimensionality reduction, and then the normalized type probability is obtained through the sigmoid activation function. Finally, the smart contract vulnerability detection result is obtained by comparing it with the set threshold.x_{final} is the output feature vector of the deep cross neural network, $\mathrm{FC}(\cdot)$ is the fully connected layer, and y is the vulnerability detection result of the model.This process is illustrated in formula (15):

$$y = \mathrm{sigmoid}(\mathrm{FC}(W_y x_{final} + b_y)) \tag{15}$$

4. Experiments

4.1. Experimental environment

The experimental environment of this article is based on the TensorFlow 2.10.0 framework, the operating system is Ubuntu 20.04, and the hardware devices are an RTX 3090 (24GB video memory) and an Intel® Xeon® Platinum 8362 CPU @ 2.80GHz (45GB memory). The development environment and the operational environment are both based on the Python 3.8 programming language, with the development tool being Pycharm 2021.2.

4.2. Data sets

This paper collected Solidity source code from Ethereum from ScrawlD[18] and SmartBugs[19]. ScrawlD provided the addresses of 5664 smart contracts and used 5 tools to generate labels for these contracts. SmartBugs used 9 tools to analyze vulnerabilities of 47587 smart contracts and provided the source code of these contracts. This paper manually selected 1926 reentrancy vulnerability datasets and 3046 timestamp vulnerability datasets from these datasets.

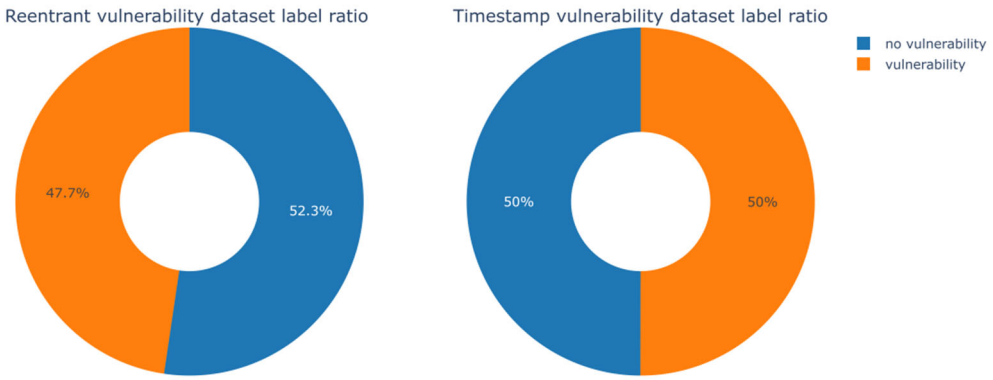

Figure 5. Dataset label ratio.

Class imbalance is a common problem in vulnerability detection training. Label imbalance can cause serious bias in the classification model, resulting in insufficient prediction ability for fewer categories. As illustrated in Figure 5, the employed dataset exhibits a favorable degree of balance, thereby facilitating a more reliable evaluation and comparison of the performance of disparate models.

4.3. Analysis of experimental results

The experimental data set is divided into training set, validation set, and test set in a ratio of 6:2:2. The learning rate is set to 0.002, the dropout is set to 0.2 to prevent overfitting, and the training rounds are 26. Figure 6 shows the changes in the loss value and accuracy of the two vulnerabilities on the training set and validation set as the training rounds increase. As the training rounds increase, the respective evaluation indicators gradually stabilize.

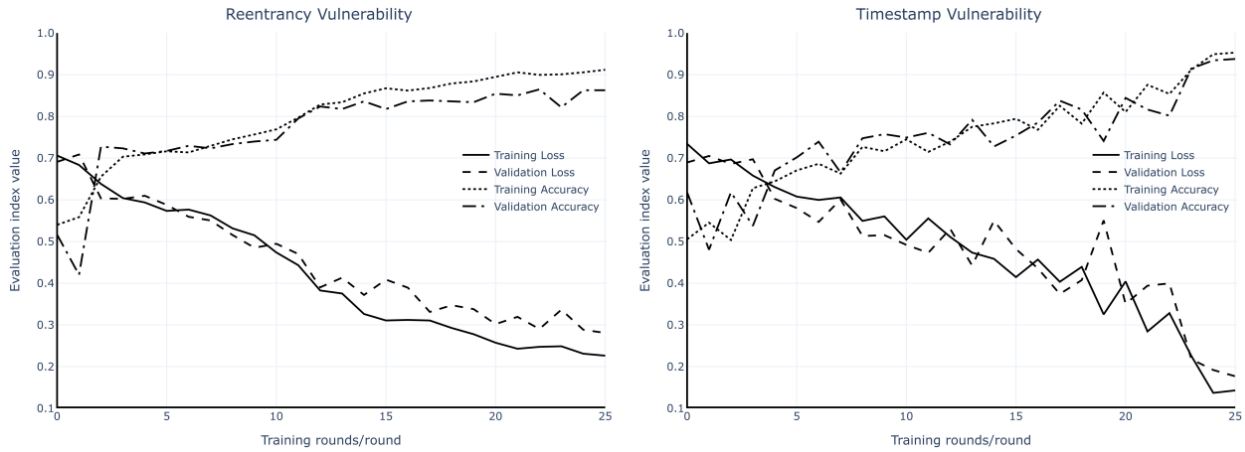

Figure 6. The evolution of model loss value and accuracy.

4.3.1. Ablation Analysis

The evaluation of this experiment employs the following indicators: accuracy, recall, precision, and $F_1-\text{score}$. In order to compare the impact of each module on the proposed method (BAMC), the smart contract vulnerability detection performance of the Bilstm-Max, Bilstm-Att, and Bi-Att-Max models is compared. The comparison results are shown in Table 1.

Table 1. Results of ablation experiments

Detection Methods	reentrancy vulnerability				timestamp vulnerability			
	Accuracy/%	Recall/%	Precision/%	$F_1-score/\%$	Accuracy/%	Recall/%	Precision/%	$F_1-score/\%$
$Bilstm-Att$	79.92	85.51	72.35	76.21	69.79	65.90	71.53	68.60
$Bilstm-Max$	86.07	81.54	83.25	82.38	84.40	87.21	82.61	84.85
$Bi-Att-Max$	87.50	86.15	82.69	84.39	86.70	87.87	85.90	86.87
BAMC	87.90	88.20	84.17	86.14	91.43	94.12	88.89	91.43

As illustrated in the table, after the introduction of the attention mechanism, the $F_1-\text{score}$ is 76.21% and 68.60% respectively. After the attention layer and the maximum pooling layer are fused, the $F_1-\text{score}$ is increased by about 2 percentage points compared with the Bilstm-Max model. After using the deep and cross network, the BAMC model reaches the highest in all aspects. Experiments show that the fusion of attention features and feature pooling features is conducive to feature extraction. After using the deep and cross network, all indicators reach the highest, indicating that the network is conducive to improving the accuracy of the model.

4.3.2. Comparison with Existing Methods

This experiment uses the evaluation indicators in the previous section to compare with existing smart contract vulnerability detection methods, namely three traditional detection methods SmartCheck,Mythril[20], Oyente and three deep learning detection methods CNN, ReChecker, TMP. The comparison results are shown in Table 2.

Table 2. Comparative experimental results

Detection Methods	reentrancy vulnerability				timestamp vulnerability			
	Accuracy/%	Recall/%	Precision/%	$F_1-score/\%$	Accuracy/%	Recall/%	Precision/%	$F_1-score/\%$
$SmartCheck$	51.26	56.11	38.49	45.66	62.31	61.16	40.27	48.56
$Mythril$	58.73	49.38	43.33	46.16	68.69	69.18	58.20	63.22
$Oyente$	57.18	44.24	52.67	48.09	55.46	48.62	42.35	45.27
CNN	68.43	71.80	65.37	66.83	78.87	87.54	71.77	76.52
$ReChecker$	79.92	82.54	71.56	76.67	69.46	81.64	65.70	72.80
TMP	83.93	81.84	72.56	76.92	85.91	84.62	79.87	82.18
BAMC	87.90	88.20	84.17	86.14	91.43	94.12	88.89	91.43

In comparison with the vulnerability detection results of the aforementioned six comparison models, the accuracy rate, recall rate, precision rate, and $F_1-\text{score}$ of this model (BAMC) have demonstrated an improvement of 3.97%,6.36%, 11.61%, and 9.22%, respectively, in terms of detecting reentrancy vulnerabilities. The accuracy rate, recall rate, precision rate, and $F_1-\text{score}$ for timestamp vulnerabilities exhibited notable improvements, with increases of 5.52%, 9.50%, 9.02%, and 9.25%, respectively. The experimental results demonstrate that, in comparison to the aforementioned control model, the smart contract vulnerability detection model proposed in this article has yielded notable enhancements across a range of indicators.

5. Conclusions and Future Work

This paper puts forth a smart contract vulnerability detection scheme based on a deep and cross neural network with feature aggregation. This scheme extracts both key features and global features from Solidity source code. This is achieved by introducing an attention mechanism and max-pooling operation. Furthermore, a deep and cross network is combined in order to learn explicit bounded-degree feature interactions, so as to detect and identify reentrancy vulnerabilities and timestamp vulnerabilities in smart contracts. The experimental results demonstrate that this method exhibits superior detection performance compared to

traditional detection tools and deep learning models. In future work, we will prioritize enhancing the overall complexity of the smart contracts within the dataset, optimizing the vulnerability detection performance of the model, and further investigating alternative vulnerability detection methods.

References

[1] Szabo N. Smart Contracts: Building Blocks for Digital Markets[J]. EXTROPY: The Journal of Transhumanist Thought, 1996, 18(2): 28-30.

[2] Dupont Q. Experiments in Algorithmic Governance: A history and ethnography of " The DAO, " a failed Decentralized Autonomous Organization[M]. 2017.

[3] Sun T, Yu W. A Formal Verification Framework for Security Issues of Blockchain Smart Contracts[J]. Electronics, 2020, 9(2):255.

[4] SharkTeam. Annual Web3 Security Report 2022[EB/OL]. (2024-01-16).https://sharkteam.org/report/analysis/20230116001A_en.pdf.

[5] Nguyen T D, Pham L H, Sun J, et al. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts[C]. International Conference on Software Engineering, 2020: 778-788.

[6] LUU L, CHU D H, OLICKEL H, et al. Making Smart Contracts Smarter[C]//ACM. 2016 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM,2016: 254-269.

[7] Antonino P, Roscoe AW. Formalising and verifying smart contracts with Solidifier: A bounded model checker for Solidity[EB/OL]. (2020-02-07)[2024-01-16]. https://arxiv.org/abs/2002.02710.

[8] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, et al. SmartCheck: static analysis of ethereum smart contracts[C]. International Conference on Software Engineering, 2018: 9-16.

[9] Yuan Z, Zhenguang L, Peng Q, Qi L, Xiang W, Qinming H, et al. Smart Contract Vulnerability Detection Using Graph Neural Network.[C]. PROCEEDINGS OF THE TWENTY-NINTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2020: 3283-3290.

[10] Nianfeng L, Yang L, Lina L, Yuying W, et al. Smart Contract Vulnerability Detection Based on Deep and Cross Network[J]. 2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA), 2022: 533-536.

[11] Peng Q, Zhenguang L, Qinming H, Roger Z, Xun W, et al. Towards Automated Reentrancy Detection For Smart Contracts Based On Sequential Models[J]. IEEE Access, 2020(8): 19685-19695.

[12] Wei W, Jingjing S, Guangquan X, Yidong L, Hao W, Chunhua S, et al. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts[J]. IEEE Transactions on Network Science and Engineering, 2021, 8(2): 1133-1144.

[13] MIKOLOV T, CHEN Kai, CORRADO G, et al. Efficient Estimation of Word Representations in Vector Space[EB/OL]. (2013-01-16)[2023-05-22]. https://arxiv.org/abs/1301.3781.

[14] Hochreiter S, Schmidhuber J. Long Short-term Memory[J]. MIT Press, 1997, 9(8): 1735-1780.

[15] Lai S, Xu L, Liu K, et al. Recurrent Convolutional Neural Networks for Text Classification[C]. AAAI Conference on Artificial Intelligence, 2015: 2267-2273.

[16] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[C]. Advances in neural information processing systems, 2017(30): 5998-6008.

[17] Wang R, Shivanna R, Cheng D Z, et al. DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems[C]. The Web Conference,2021: 1785-1797.

[18] Yashavant C S, Kumar S, Karkare A. ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities[EB/OL]. (2022-02-25)[2024-01-18]. https://arxiv.org/abs/2202.11409.

[19] Durieux T, Ferreira J F, Abreu R, Cruz P, et al. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts[C]. International Conference on SoftwareEngineering, 2020: 530-541.

[20] Mythril:security analysis tool for EVM bytecode[EB/OL]. [2023-05-01]. https://github.com/ConsenSys/mythril.