Journal of Combinatorial Mathematics
and Combinatorial Computing
www.combinatorialpress.com/jcmcc

Combinatorial Press

Optimizing Software Supply Chain Vulnerability Mining and Remediation Paths Using Deep Learning Techniques

Chenyu Liu[1], Jianheng Shi[1], Shubin Yuan[1,]✉

[1] *CNOOC Information Technology Co., Ltd., Beijing Branch, Beijing, 102209, China*

ABSTRACT

This paper constructs an overall framework for vulnerability mining, covering the whole process from code collection to vulnerability remediation. The word vector technique is used to transform code fragments into vector form, thus preserving the semantic information of the code. A vulnerability mining system based on semantic graph of source code is further designed, which generates a semantic graph of code by constructing an abstract syntax tree (SAT), and analyzes the semantic graph by using graph neural network to accurately locate potential vulnerabilities. At the same time, a vulnerability repair method based on thought chain is proposed. The results show that the model in this paper can accurately mine the vulnerabilities of web service software, and it consumes short latency and has strong stability. The results of web service software vulnerability detection show that the accuracy rate of the model always stays above 85% under different network structures. In addition, this paper obtains that the integration degree centrality measure and 60 iteration rounds have the best effect on the detection of vulnerabilities of the model. Finally, the vulnerability repair experiments show that at Beams=15, the model in this paper repairs each vulnerability function with a PPP metric of 61.52% and an average time of 3.168 seconds, which is the best for vulnerability repair.

Keywords: Vulnerability Mining, Word Vector Technique, Source Code Semantic Map, SAT, Chain of Thought Vulnerability Repair Method

1. Introduction

In today's digitized business world, software has become a core component of enterprise operations and is ubiquitous from internal business process management to interaction with customers [1-2]. However, as software applications become more widespread, security and reliability issues in the software supply chain are becoming more prominent [3]. The software supply chain is like a complex production line involving multiple links such as software development, integration, distribution, and maintenance, etc. If there is a security breach in any of these links, it may cause serious impacts on the

✉ Corresponding author.
E-mail address: 18810525396@163.com (S. Yuan).
Received 20 February 2024; Revised 15 May 2024; Accepted 20 November 2024; Published Online 16 April 2025.
DOI: 10.61091/jcmcc127b-317

organization, including data leakage, business interruption, and reputation damage [4-7]. With the rapid development of the software industry, the software supply chain has become more and more complex and diversified, and the complex software supply chain introduces a series of security problems, which leads to an increasing difficulty in the overall security protection of information systems [8-10]. Therefore, timely repair to ensure the security and reliability of the software supply chain has become an important challenge that enterprises must face.

Vulnerabilities against the software supply chain often utilize the inherent security holes in the system, or pre-built software backdoors to carry out attack activities [11]. Through the network chain structure formed by the software supply chain will spread the attack effect downstream to all the participants in the supply chain, through the excavation of vulnerabilities in the software supply chain after the targeted repair, the common repair steps are generally to confirm the scope of the vulnerability, vulnerability analysis, the development of a repair plan, the implementation of repair, and testing [12-15]. With the development of artificial intelligence, deep learning technology is widely used in optimizing software supply chain vulnerability mining and repair [16-17]. Deep learning is a branch of machine learning that is based on artificial neural networks and uses a large amount of data and computational resources for training and optimization [18-19]. In the field of software supply chain, it is able to utilize advanced technologies and algorithms by analyzing and processing big data and deep learning methods so as to realize vulnerability mining and repairing in software supply chain [20-22].

In this study, a general framework for software vulnerability checking is designed. Word2Vec technique is used to convert the code into a sequence of word vectors, and the model is trained by constructing positive and negative samples to realize the automatic detection of software vulnerabilities. In order to be able to retain deeper code semantic information in a more direct way during the preprocessing process, a vulnerability mining system SGVDNet, which can extract vulnerability features from graphical representations of source code for vulnerability detection, is designed in conjunction with the graph neural network model, and a vulnerability detection scheme based on the semantic graph of source code is proposed. Aiming at the problem that the generation space of the fixing program is large and the quality of the generated fixing program is low, a vulnerability fixing method based on the chain of thought is proposed to predict the locations with higher probability of generating vulnerabilities. Combined with the prediction results, the repair program is generated more accurately. Finally, the vulnerability detection and repair effect of the model is analyzed.

2. Software supply chain vulnerability mining and remediation techniques

2.1. General Framework of Vulnerability Mining

The overall theoretical framework of deep learning-based vulnerability mining technology proposed in this paper mainly includes data preprocessing, code characterization, deep learning, analysis and optimization. Data preprocessing is the foundation of vulnerability mining, its main task is to extract useful information from a large amount of source code, the specific steps include code collection, code cleaning, code labeling and feature extraction. Code characterization can convert the source code into a form that can be processed by the deep learning model, this paper adopts the code characterization method based on word vectors, which can represent the code statements or code fragments as high-dimensional vectors. In the deep learning part, this paper adopts the RNN model, through the training of the preprocessed code data, the model can identify the code vulnerability. Finally, the vulnerabilities detected by the deep learning model need to be verified and the false positives and omissions analyzed to improve the practical application of the model.

2.2. Word Vector Code and Representation

Prior to code detection, this paper uses Word2Vec method [23] to extract word vectors to characterize software code.The core idea is to map words with similar semantics to similar vector space locations based on contextual context. The method mainly consists of 2 models: the continuous bag of words (CBOW) model, and the Skip-gram model. This paper focuses on the Skip-gram model.

Suppose given a word sequence $w_1, w_2, \cdots, w_T$, the objective function maximization method is as follows:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\ne 0}\log P(w_{t+j}\mid w_t) \tag{1}$$

where w_t denotes the current word, w_{t+j} denotes the context word with window size c, and $P(w_{t+j}\mid w_t)$ denotes the probability of predicting the word w_{t+j} given the word w_t.

To compute the conditional probability $P\left(w_{t+j}\mid w_t\right)$, the method is modeled using the Softmax function, i.e.:

$$P\left(w_{t+j}\mid w_t\right)=\frac{\exp(v'_{w_{t+j}}v_{w_t})}{\sum_{w=1}^{W}\exp(v'_w v_{w_t})}\sqrt{a^2+b^2} \tag{2}$$

where v_{w_t} is the word vector representation (input vector) of the word w_t, $v'_{w_{t+j}}$ is the word vector representation of the word w_{t+j}, and W is the vocabulary size.

In order to improve the computational efficiency, this paper approximates the optimized objective function by negative sampling. The basic idea of this method is to transform the multi-classification problem into a binary classification problem and train the model by constructing positive and negative samples. The objective function can be expressed as:

$$\log\sigma(v'_{w_{t+j}}v_{w_t})+\sum_{i=1}^{k}\mathrm{E}_{w_i\sim P_n(w)}\log\sigma\left(-v'_{w_i}v_{w_t}\right) \tag{3}$$

where $\sigma(x)=\dfrac{1}{1+\exp(-x)}$ is the Sigmoid function; k is the number of negative samples; and $P_n(w)$ is the noise distribution.

Next, the model parameters are updated by stochastic gradient descent (SGD). The update rules for each training sample are as follows.

1) Positive sample update:

$$v_{w_t}\leftarrow v_{w_t}+\eta[1-\sigma(v'_{w_{t+j}}v_{w_t})]v'_{w_{t+j}} \tag{4}$$

$$v'_{w_{t+j}}\leftarrow v'_{w_{t+j}}+\eta[1-\sigma(v'_{w_{t+j}}v_{w_t})]v_{w_t} \tag{5}$$

2) Negative sample update:

$$v_{w_t}\leftarrow v_{w_t}-\eta\sigma(v'_{w_i}v_{w_t})v'_{w_i} \tag{6}$$

$$v'_{w_i}\leftarrow v'_{w_i}-\eta\sigma(v'_{w_i}v_{w_t})v_{w_t} \tag{7}$$

where η is the learning rate.

Through the above process, the Skip-gram model is able to effectively learn the low-dimensional vector representation of words and provide effective support for subsequent vulnerability mining.

2.3. Vulnerability Mining System Design Based on Source Code Semantic Map

2.3.1. Workflow. Vulnerability mining system [24-25] SGVDNet aims at predicting the presence of a security vulnerability for each function from the source code input.SGVDNet introduces learnable parameters for the different types of edges, which are used to obtain a matrix of vectors that reflect the semantic features of the code. The final result prediction stage is consistent with VDCNet, where after a dense layer classifier, the final output is a probability distribution $P_t\{p_1, p_2, \ldots, p_t\}$ for t generic vulnerability types. Similarly, SGVDNet takes $P_m = \max\{p_1, p_2, \ldots, p_t\}$ as the vulnerability prediction result of the model.

In the whole workflow, the source code is mapped into the graph code representation in terms of functions, and the system maps the AST nodes of each function and the various semantic relationships between the nodes as nodes with multiple directed edges in the graph, respectively.SGVDNet sends the whole graph features of the functions as inputs to a classifier, thus realizing the vulnerability detection of the source code at a fine-grained level of the function.

2.3.2. AST-based code semantic graph generation. This subsection demonstrates how to map the source code into a code semantic graph assembled with edges consisting of AST nodes and various semantic relationships between nodes.

1) Source Code Preprocessing. In order to be able to perform the vulnerability detection task at a function-level granularity, SGVDNet performs data cleaning and function segmentation on the raw inputs that are exactly the same as VDCNet.

2) Code Semantic Graph Definition. In order to generate code representations based on graphical data structures, this section uses the open-source vulnerability analysis tool Joern as the front-end parsing tool for source code. ASTs generated using Joern can be saved in a JSON-like structure, which can facilitate the extraction of AST nodes and the construction of adjacencies between nodes.

This scheme defines a code semantic graph (CSG), which is a graphical structural representation of source code: for a given source code file $P = \{FT_1, FT_2, \ldots, FT_m\}$, the CSG of any function FT_i is defined as $G_S^i\left(V_S^i, E_S^i\right)$, where $V_S^i = V_A^i$ represents the set of points of the CSG, which corresponds to the set of leaf nodes in the AST graph of FT_i; and $E_S^i = E_A^i \cup E_C^i \cup E_D^i \cup E_L^i$, where E_A^i is the AST parent-child relationship edge of FT_i, E_C^i is the control dependency edge in the control flow graph of FT_i, E_D^i is the data flow graph of FT_i, the data dependency edges, and E_L^i is the leaf node ordering edges additionally added to the AST in this scheme.

3) Code Semantic Graph Generation. First, use Joern to generate AST, CFG, and DFG for each function in the source file, respectively, and then use AST to initialize the nodes V_A and edges E_A of CSG. Next, traverse all nodes in V_S in depth-first order.

2.3.3. Vulnerability Detection Model. This scheme uses a graph neural network model B^g to process CSGs, whose feature extraction layer is a GGNN-structured network model.

1) Input layer initialization. In the previous subsection, the source code is modeled as a CSG that can characterize the syntactic structure and semantic information of the code, which is composed of an AST node V_A and four kinds of directed edges E_s. In order to convert $G_s\left(V_s, E_s\right)$ into an input form that can be computed by B^g, it is necessary to initialize an eigenvector for each node $v_i \in V_s$, mapping it to the vector space.

In this paper, we use dictionary lookup to initialize CSG nodes: first map V_s to a sequence of tokens, then load the token embedding generated by B^e for each of the words during pre-training, and find the word vector of v_i i.e., $e_i^{token} \in \square^{192}$.

2) Feature extraction and vulnerability classification. For the node v, its hidden layer state h_v is first populated from the embedding vector x_v to the predefined dimension d by complementary zeros before the start of the first time step of the GGNN, Eq:

$$h_v^0 = pad\left(x_v, d\right) \tag{8}$$

During the aggregation phase at the moment t, each node communicates its current state h_v^{t-1} to all its neighboring nodes, using the MLP as a messaging function in this process:

$$M_v^t = \sum_{v' \in N(v)} MLP\left(h_v^{t-1}, h_{v'}^{t-1}, \left(e_{vv'}\right)\right), \forall \left(v, v'\right) \in A_v \tag{9}$$

where A_v is the edge relation corresponding to the two columns of the outgoing edge A_v^{out} and the incoming edge A_v^{in} of the node v, and $N(v)$ is the set of all neighboring nodes of v, and e_{vv}, denotes all of its neighbors. h_v^{t-1} is the state vector of node v in the previous time step, and M_v^t is the information about the neighboring nodes that node v has aggregated to at moment t.

At each time step, each node needs to update its state vector based on the messages obtained from the aggregation, a process that relies on the GRU units in the GGNN. In order to obtain the state vector h_v^t of the node v at the moment t, it is necessary to take into account both the state vector h_v^{t-1} of the previous moment and M_v^t obtained from the aggregation at this moment. The computation of the update gate z_v^t and the reset gate r_v^t is first accomplished using the product of h_v^{t-1} and M_v^t and the weight matrices U and W, respectively, in the following process:

$$z_v^t = \sigma\left(U^Z h_v^{t-1} + W^Z M_v^t\right) \tag{10}$$

$$r_v^t = \sigma\left(U^R h_v^{t-1} + W^R M_v^t\right) \tag{11}$$

After obtaining the states of the two gates, the state vector h_v^t of the node v is then computed, i.e:

$$\tilde{h}_v^t = \tanh\left(U^h \left(r_v^t \odot h_v^{t-1}\right) + W^h M_v^t\right) \tag{12}$$

$$h_v^t = z_v^t \odot \tilde{h}_v^t + \left(1 - z_v^t\right) \odot h_v^{t-1} \tag{13}$$

The computation of the candidate state $\tilde{h}_v^t$ is activated by the $\tan h$ function, and the reset gate and update gate are involved in the computation of $\tilde{h}_v^t$ and h_v^t, respectively, where $\odot$ denotes element-by-element multiplication. For any node, its final feature vector is obtained after repeating to the time step T times preset for GGNN.

The computational process of GGNN is implemented based on the Pytorch Geometric library, and the features of the same node in different subgraphs are aggregated based on the node weights generated by MLP learning before the end of each time step to ensure that the model is able to learn semantic information on different edges according to their importance:

$$h_v = \sum_{l \in N} \sigma\left(MLP\left(h_v^l\right) h_v^l, v \in V_S\right) \tag{14}$$

where h_v^l represents the state vector of node v in the l th layer, h_v is the state vector obtained by aggregating the hidden layer states of node v in all the layers of the CSG, and h_v^l is summed up according to the weights weighted by the weights computed by the MLP, and N has the value of 4, representing the of the four edge relationships.

After T time steps of aggregation and updating, the extraction method of SGVDNet for whole graph features follows the pattern of VDCNet, i.e., performs global maximum pooling sampling of feature vectors of all nodes in the graph. Next, the whole graph features are input to the fully connected layer computation and activated using the Softmax function to obtain the final vulnerability prediction results, and the training process uses the cross-entropy loss function. The whole process is formulated as follows:

$$O_{\Pr ed} = \max\left(Soft\max\left(\left(\max_{v=1}^{|V_S|}\left(h_v[j]\right)\right)W^f + b\right)\right) \tag{15}$$

2.4. Vulnerability Remediation Methods Based on Chain of Thought

This paper proposes a vulnerability repair method based on chain of thought, which is mainly composed of data preprocessing phase, model training phase and vulnerability repair phase.

2.4.1. Data Preprocessing and Data Representation. Data preprocessing: For a vulnerability fix submission record that exists in the dataset, the pre-modification and post-modification codes are extracted by searching the historical fix records. Then use tree-sitter to remove all the comments in the original vulnerability program and the fix program. Using tree-sitter to eliminate all the functions that do not produce changes, the functions that produce changes are respectively divided into words using the BPE segmentation algorithm to obtain the vulnerability program and the repair program. Finally, the difflib tool is utilized to compare the two sequences and generate the fix location information.

DATA REPRESENTATION: In terms of inputs to the model, this paper follows the previous history-driven vulnerability-based automated remediation approach by using the tagged vulnerability program as input. In this case, two special tokens $\langle StartLoc \rangle$ and $\langle EndLoc \rangle$ are used to mark the 1st line of defect code in the vulnerability program. For defects that appear on multiple lines, only the first line of defect code is also marked.

Code Segmentation: for the input vulnerability program with fixes, the input vulnerability program is firstly segmented using the BPE algorithm.The BPE algorithm will divide the longer code tokens into smaller but more common sub-token sequences.These stems help to build better word embedding models and language models, and have been widely used in language models.

2.4.2. Model training. The model architecture of this paper follows the T5 model [26], which is an encoder-decoder architecture. Both decoder and encoder consist of 12 layers, each containing 12 attention header transformer layers. This architecture has been applied to many tasks and achieved state-of-the-art results.

Embedding Layer: the embedding layer consists of two parts: the code embedding layer and the location coding layer. Assuming that for the input vulnerability program C, a sequence of token streams $(t_i,\ldots,t_n)$ is obtained after word splitting, and then the semantic vector x_i corresponding to the ith token is searched for in the word embedding matrix E_w to obtain the embedding matrix of the vulnerability program C, $X = Concat(x_1,\ldots,x_n)$, and $x \in \mathbb{R}^{n \times d_x}$. The specific process can be expressed as follows:

$$x_i = lookup(t_i, E_w) \tag{16}$$

For the positional coding layer, the positional coding layer is used to capture the relative positional relationship between any two elements x_i and x_j in the embedding matrix X, which encodes the relative positional relationship from x_i to x_j as two vectors $a_{ij}^K, a_{ij}^V \in \mathbb{R}^{d^\omega}$, and the specific process can be expressed as follows:

$$\begin{cases} a_{ij}^K = w_{clip(j-i,k)}^K \\ a_{ij}^V = w_{clip(j-i,k)}^V \\ clip(x,k) = \max(-k, \min(k,x)) \end{cases} \tag{17}$$

where the Clip function serves to crop the relative position between two elements whose distance is too large, i.e., if the relative position between two elements is greater than k, the relative position between them is considered to be k; the relative position encoding matrices $w^K = (w_{-k}^K,\ldots,w_k^K)$, and $w^V = (w_{-k}^V,\ldots,w_k^V)$ is the learnable weight matrix, where w_i^K, $w_i^V \in \mathbb{R}^{d_a}$, $d_a = 64$.

Encoder: A linear superposition of 12 structurally identical encoder layers used to encode the input data into a dense representation while preserving as much as possible some features of the original

data.

For a single attention head with relative position encoding, the receiver matrix $X = Concat(x_1,\ldots,x_n)$, and $x \in \mathbb{R}^{n\times d_x}$ as inputs are utilized with 3 linear transformation matrices $W^Q, W^K, W^V \in \mathbb{R}^{d_x\times d_z}$ to project the input variables to three matrices belonging to three subspaces, denoted as Q, K, and V, respectively, each responding to different Each subspace responds to different hidden features. The specific computational procedure can be expressed as follows:

$$Attention(Q,K,V) = Soft\max\left(\frac{Q\left(K+P^K\right)^T}{\sqrt{d_z}}\right)\left(V+P^V\right) \tag{18}$$

where Q, K and V, P^K, P^V are calculated as follows:

$$\begin{cases} Q = XW^Q, K = XW^K, V = XW^V \\ P^K = Concat(p_1^K,\cdots,p_n^K), p_n^K = \sum_{j=1}^{n} a_{ij}^K \\ P^V = Concat(p_1^V,\cdots,p_n^V), p_n^V = \sum_{j=1}^{n} a_{ij}^V \end{cases} \tag{19}$$

After all self-attention heads are computed, the results are connected and fed into the feedforward neural network. Therefore, the specific process of the multi-head self-attention mechanism can be expressed as follows:

$$\begin{cases} head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \\ MultiHead\left(Q,K,V\right) = Concat(head_1,\cdots,head_h)W^O \end{cases} \tag{20}$$

where $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_x\times d_z}$ is the weight matrix for mapping the inputs in the ith attention head; $W^O \in \mathbb{R}^{hd_z\times d_x}$ is used to linearly map the merged vectors in order to obtain the output, where $h=12$, the number of subspaces, and the rest of the parameter values are $d_z = d_x / h = 64$.

Finally, the input is processed by a layer of feed-forward neural network, which aims to make the model more flexible by adding nonlinear transformations that can better capture the relationships between the elements of the individual input sequences, thus fully utilizing the information within the sequences. The specific process can be expressed as follows:

$$FFN\left(x\right) = \max\left(0, xW_1 + b_1\right)W_2 + b_2 \tag{21}$$

where W_1, $W_2 \in \mathbb{R}^{d_s\times d_s}$ are the weight matrices for transforming the inputs, and b_1, b_2 are the offsets, and the weight matrices of the feedforward neural networks in the different encoder layers have independent parameters . The encoder consists of a linear superposition of 12 structurally identical encoder layers, where the output of the previous layer is the input of the next layer. Finally, the output of the last encoder layer is extracted as the extracted hidden layer state $E \in \mathbb{R}^{n\times d_s}$.

Decoder: After obtaining the hidden layer state E extracted by the encoder, the model decodes the feature using a decoder. The decoder consists of a linear superposition of 12 identical decoding layers, with the input to layer 1 being the hidden layer state E and the vector matrix $O \in \mathbb{R}^{n\times d_s}$ of the expected outputs after the code embedding layer. The vector matrix O is first passed through the masked multi-head self-attention layer, and then the same hidden layer state E, is fed into the multi-head self-attention layer, and the output U_1 of the first layer is obtained after the feed-forward neural network. The specific process can be expressed as:

$$\begin{cases} U_1 = FFN\left(MultiHead\left(E,E,O'\right)\right) \\ O' = MaskMultiHead\left(O,O,O\right) \end{cases} \tag{22}$$

After this, the input of each i-layer encoder is the output of the $i-1$-layer encoder. The specific process can be expressed as follows:

$$\begin{cases} U_i = FFN(MultiHead(E,E,U'_{i-1})) \\ U'_{i-1} = MaskMultiHead(U_{i-1},U_{i-1},U_{i-1}) \end{cases} \tag{23}$$

The output of the last layer accesses a fully connected layer that projects the output into a vector with dimensions equal to the size of the word list, which is then transformed into a probability distribution over the word list by a Softmax function to generate the final token probability distribution matrix. The specific process can be expressed as:

$$Q = Soft\max\left(U_{-1}W^{vocab}\right) \tag{24}$$

Where $W^{vocab} \in \square^{d_s \times d_{vocab}}$, d_{vocab} is the size of the word list.

Loss computation: in this paper, we use the cross loss function to train the model, and the specific process of loss function computation can be expressed as:

$$Loss = -\sum_{i=1}^{n} P\left(t_i\right)\log\left(Q\left(t_i\right)\right) \tag{25}$$

where t_i is the ith element in the stream of tokens generated by the model, P is the probability distribution of tokens in the ideal case, and Q is the probability distribution of tokens generated by the model.

2.4.3. Vulnerability remediation. After the model is trained, it can be used to generate candidate repair programs. Specifically, the original vulnerability program is extracted first, and after removing all annotations, the BPE algorithm is used to obtain the vulnerability program. After that, the vulnerability program is inputted into the model, and the model extracts and decodes the features of the sequence to generate the probability distribution of the tokens that constitute the repair program. In the process of generating the repair program, we utilize the cluster search strategy to select the k candidate repair programs with the highest probability for output, i.e., to obtain the candidate repair programs predicted by the model.

3. Analysis of the effectiveness of software supply chain vulnerability mining and remediation

3.1. Simulation experiment and result analysis

In order to verify the performance of the application of this paper's method in realizing the vulnerability mining of web service software, simulation tests are carried out, combining MATLAB and C++ for the vulnerability mining analysis of web service software, and the distribution intervals of the parameters of the reliability indexes of the web service software are set as [0, 0.3], [0.31, 0.5] and [0.51, 0.8]. In this paper, 2 traditional testing methods, named VDM1 and VDM2, are used to simulate and test their performance with the model of this paper.

The spectral distribution of the vulnerability detection output is shown in Fig. 1. It is analyzed that using the method of this paper can effectively realize web service software vulnerability mining, and the reliability of the spectral distribution is good.

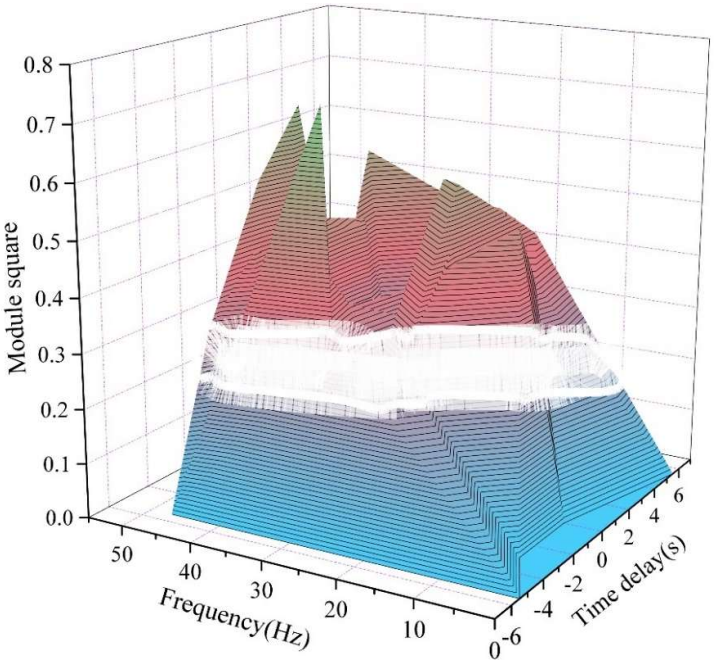

Fig. 1. Vulnerability detection output spectrum distribution

The results of the comparison of the time delay of the vulnerability detection output are shown in Figure 2. Analysis has shown that this paper's method for web service software vulnerability mining time delay than the comparison model is significantly shorter, the fluctuation interval is smaller, the stronger the stability, and with the increase in the number of tests, the time delay is infinitely close to 0, which can be seen in this paper's model of the vulnerability detection of the output of the higher efficiency.

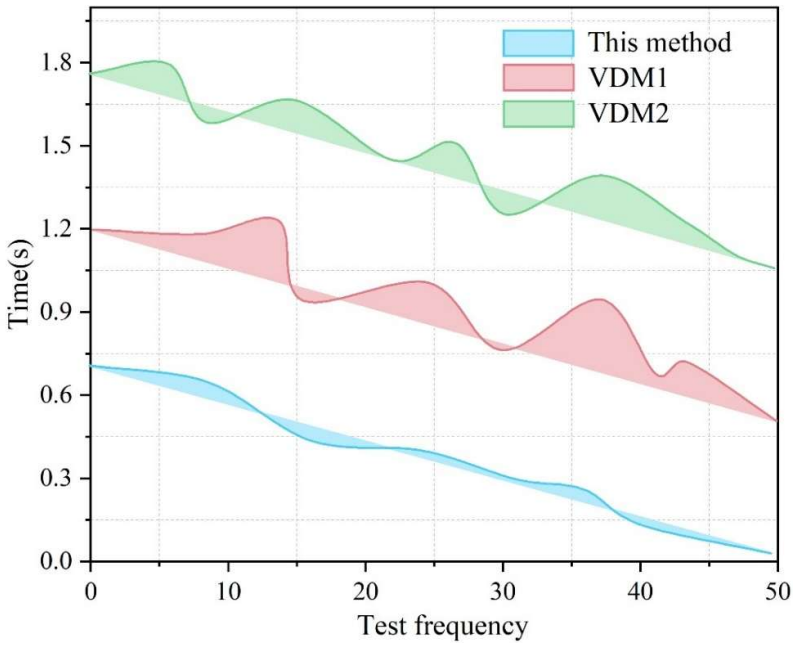

Fig. 2. The delay contrast results of the leak test output

On the basis of the above experiments, the vulnerability mining accuracy rates of the three methods are compared. The results of vulnerability mining accuracy comparison are shown in Figure 3. The analysis shows that the vulnerability mining accuracy of VDM1 varies from 62.27% to 71.83%, and the

vulnerability mining accuracy of VDM2 varies from 59.47% to 71.44%, while the vulnerability mining accuracy of the research method always remains above 95%, indicating that the method can realize the accurate mining of vulnerabilities in web service software.

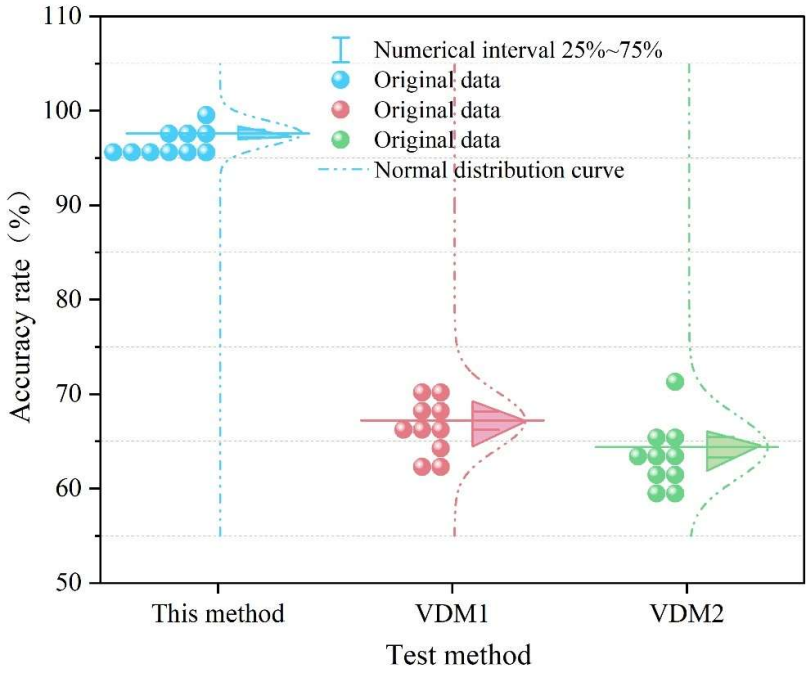

Fig. 3. The accuracy comparison results of vulnerability mining

3.2. Analysis of Vulnerability Detection Effectiveness of SGVDNet System

3.2.1. Experimental data set. In this paper, we first collected the SARD vulnerability dataset. It is a project maintained by the National Institute of Standards and Technology (NIST) and contains 118 CWE vulnerability types. Each sample in the dataset is labeled by its function name with a suffix string of "good" or "bad", so it is easy to use this suffix information to generate a label for each sample. In this paper, the experiments focus on identifying vulnerabilities in C/C++ source code. Therefore only C/C++ data from the SARD dataset is used, which contains 12,345 vulnerable functions and 21,583 benign functions without vulnerabilities.

The dataset is divided into training, testing and validation sets according to the ratio of 8:1:1 respectively. In this paper, the performance of vulnerability detection is measured in terms of precision, recall, accuracy and F1_score.

In order to investigate the optimal network structure of SGVDNet and the optimal node centrality metric, this paper first explores the impact of different network structures on the detection effect, and explores the impact of four different types of node centrality metrics on vulnerability detection. Finally, the optimal network structure and the optimal centrality measure are compared with various vulnerability detection methods to prove the effectiveness of the proposed method.

3.2.2. Experimental results and analysis:

1) Experimental results for different network structures. In this paper, the experiments explore the vulnerability detection of the model under the same dataset by classifying various network structures according to whether to use edge attributes including control flow and data flow information, whether to use SAT structure, whether to use k-subgraph structure or k-subtree structure, whether to use Layer norm or Batch norm, and which type of GNN to use. The effect of vulnerability detection model is examined. The Layer norm refers to the normalization of a batch of data during the training process,

while the Batch norm refers to the normalization of each row of the samples in each batch.

The different network structure vulnerability detection results are shown in Table 1. From the table, it can be seen that comparing Structure1 and Structure2, although the results of using k-subgraph structure extractor are better, the difference with k-subtree is not significant. k-subgraph takes more time in the subgraph extraction process, and this slight advantage is not enough to offset the time it takes, so the real-time nature of the detection is considered. In this paper, we finally use only k-subtree as the structure extractor in the SGVDNET network model. Comparing Structure1 with Structure3, the accuracy, precision, recall and F1_score of the detection results are improved substantially after introducing the edge information representing data dependency and control dependency into the model training, this is because some vulnerabilities tend to be introduced by the data flow or the transfer of control information, so discarding such information is not desirable. The edge attributes must be taken into account in the network model implemented in this paper to utilize the data dependency and control dependency relationships.

Comparing Structure1 with Structure4, the introduction of SAT attention structure can effectively improve the vulnerability detection effect, this is because SAT not only retains the data dependency and control dependency information, but also retains the code line order information as well by means of positional encoding, and at the same time avoids the excessive smoothing phenomenon that exists in general GNNs. Comparing Structure1 with Structure5, compared with batchnorm, the use of layernorm reduces the model's accuracy and precision performance to a certain extent, but the model's recall and F1_score gain a certain degree of enhancement, but this paper is more from the accuracy, and in the final network model, the use of the batchnorm normalization operation. Comparing Structure1, Structure6, and Structure7, the highest accuracy, precision, and F1_score were obtained using PNA, and the highest recall was obtained using GCN. This is also precisely because PNA can better understand the information in the graph structure and retains more information about the neighbors, all things considered, PNA is chosen as the final SGVDNet model structure used in this paper.

Table 1. Different network structure vulnerability detection results

Network structure number	Accuracy rate	Accuracy rate	Recall rate	F1_score
Structure1	90.25	84.84	90.21	87.48
Structure2	90.3	84.94	90.23	87.51
Structure3	87.25	82.41	84.76	83.63
Structure4	88.86	85.77	84.64	85.22
Structure5	89.48	81.15	95.47	87.67
Structure6	89.57	82.84	91.72	87.05
Structure7	88.96	84.51	87.15	85.89

2) Experimental results of incorporating different node centrality metrics. In this paper, by introducing node centrality, we aim to express more information in the graph structure in the feature vectors, and in order to explore the best node centrality metrics, the model without embedding node centrality (NUNC) is compared with the model after embedding degree centrality (UC), Katz centrality (KC), proximity centrality (UP), and harmonic centrality (HC), and comparing them in terms of detecting the difference in effectiveness. In this paper, 10 tests are conducted and then the mean values are obtained for comparison as follows.

The vulnerability detection effects of incorporating different node centrality metrics are shown in Figure 4. From the results in the figure, it can be seen that in addition to the use of proximity centrality reduces the accuracy of vulnerability detection, the accuracy of the other three centrality metrics in the vulnerability detection task is higher than the original model without centrality, especially the use of degree centrality for the model to improve the performance of the detection of the largest. The possible reasons for this are: proximity centrality measures the average of the sum of the shortest path lengths from one point to all other points, whereas the PDG graph used in this paper's method is

directional, which will include some unreachable points, which is equivalent to the introduction of more noise for the model as a whole, leading to a decrease in the final detection effect.

The vulnerability pattern features that can be extracted for nodes that are not directly connected are extremely limited, so using the simplest metric of centrality instead helps the model the most in terms of detection accuracy.

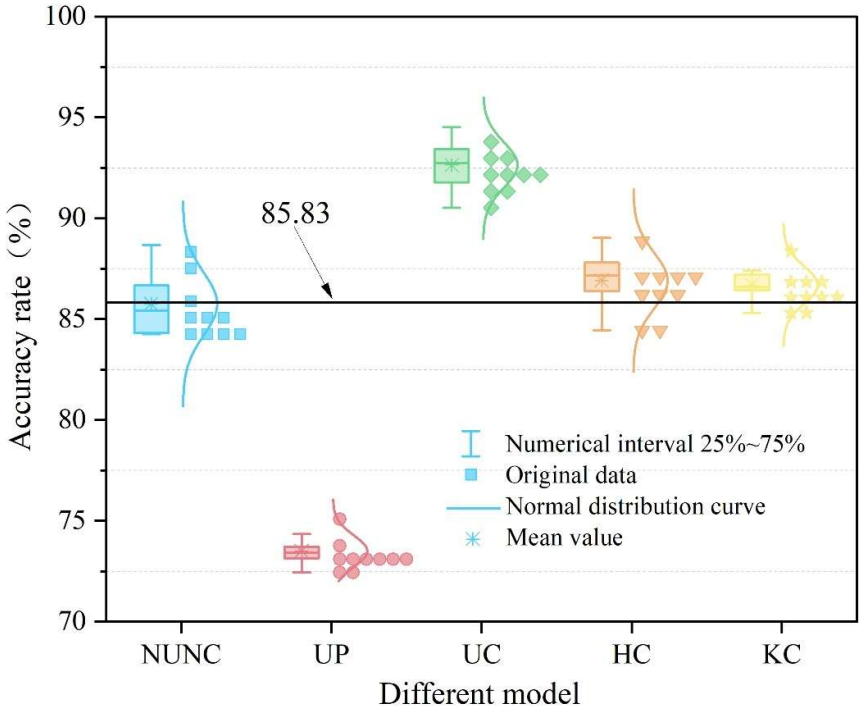

Fig. 4. The vulnerability detection effect of the node center metric method

3) Setting of iteration rounds in the model training process. In this paper, the number of iteration rounds set during the training process of the SGVDNet model is 100, in order to explore whether the iteration rounds are set reasonably, the results of the iteration rounds and the loss relationship are shown in Figure 5. From the figure, it can be seen that the value of the loss function decreases to the lowest value (0.035) after 54 rounds and keeps converging, in order to prevent the phenomenon of overfitting of the model and the time overhead caused by too many iteration rounds, this paper sets the number of iteration rounds of the selected dataset to 60 during the training process, which is more reasonable.

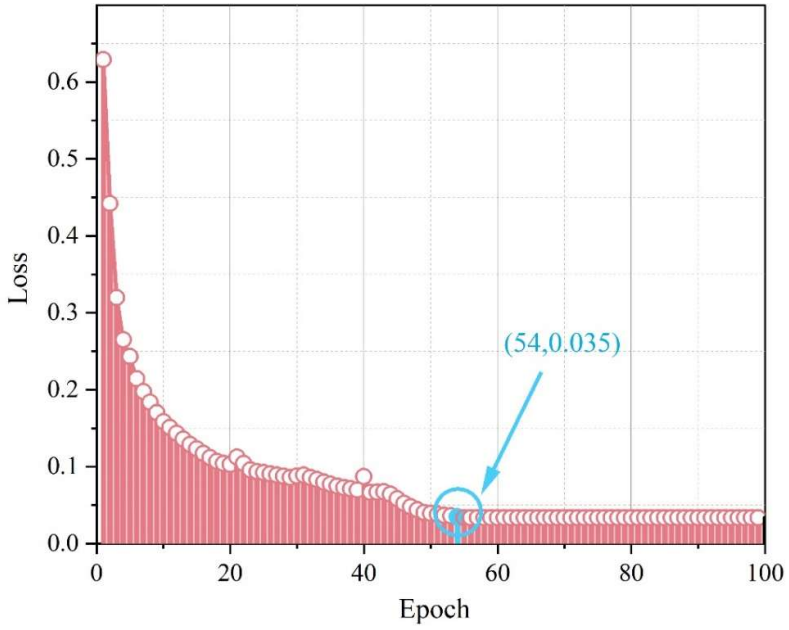

Fig. 5. Iterative rotation and loss relationship results

3.3. Analysis of the effectiveness of vulnerability remediation in the system

3.3.1. Experimental setup. This paper uses the same real-world C/C++ vulnerability fixing dataset that is used to evaluate existing automated vulnerability fixing methods. The dataset consists of two existing datasets, Big-Vul and CVEfixes, covering 8203 vulnerability fixes for 1582 large open source software projects and over 150 different types of CWE categories. For each pair of vulnerability functions and their fixes. In the vulnerability function, each vulnerability region is identified by a special token, and de-duplication is performed on the dataset to obtain the number of training, validation and test sets of 4200, 600 and 600 respectively, totaling 5400.

Consistent parameter settings are used to train the vulnerability concern model and vulnerability repair model: batch size is 6, training rounds are 50, learning rate is 0.001, and the input sequence length is set to 524. For the vulnerability repair model, the output sequence length is set to 396. In order to determine the appropriate output length, the statistical results of this study on the number of tokens within a patch are shown in Figure 6. After analysis, this study selects 384 tokens as the output length of the remediation model, which is able to cover more than 99% of the patches in the dataset, ensuring high coverage, accelerating the generation process, reducing resource consumption, and effectively preventing the problem of infinite generation. This shorter output length significantly improves the efficiency of vulnerability remediation and avoids the long time needed to directly generate the entire code snippet. In addition, the T5 model is chosen as the infrastructure of this paper's approach, which brings better performance and results for handling code-related tasks thanks to its excellent generality, moderate model parameter size and ability to support multiple tasks.

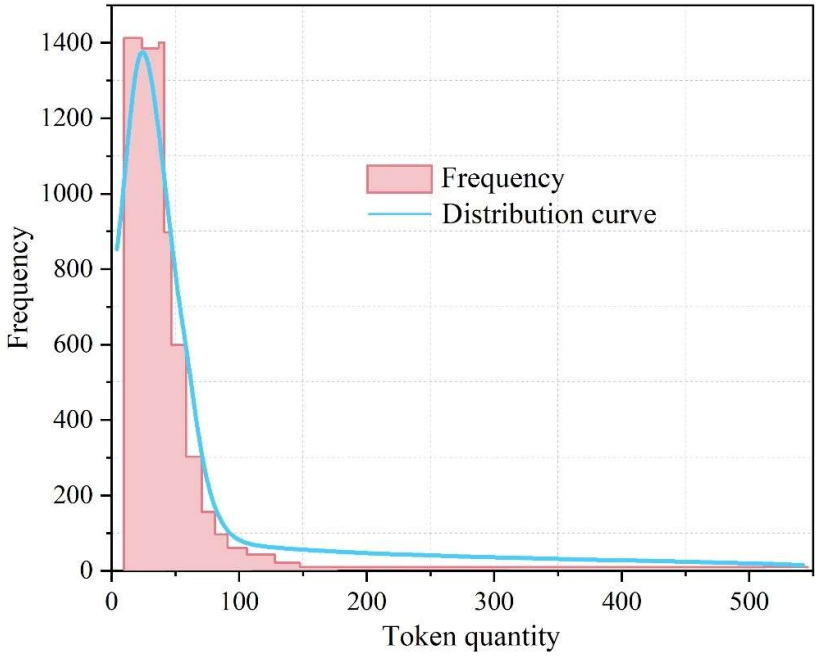

Fig. 6. A map of the number of tokens in the patch

3.3.2. Analysis of experimental results. In order to evaluate the performance of the chain-of-thought method (CRM) on the task of automatic vulnerability remediation, the perfect percentage prediction (PPP) was selected as the evaluation metric in this study. In automatic vulnerability remediation, if any of the remediation patches generated by the chain-of-thinking method using the bundle search algorithm perfectly matches the target sequence in the dataset, the prediction is deemed successful. Accordingly, the PPP of the test set is derived by dividing the number of correctly predicted samples by the total number of samples in the test set. The chain of thought approach methodology proposed in this paper was analyzed in comparison with the following baseline methods:

VulRepair: an automated vulnerability remediation approach based on the large language model T5.

VRepair: an automated vulnerability repair method that combines transformer modeling and migration learning techniques to fine-tune to the vulnerability repair task after training with a large number of bug fixes.

TFix: a large-scale automatic vulnerability repair model that relies on T5 and is pre-trained with a natural language corpus.

CURE: An automated vulnerability repair method that implements code-aware strategies to enhance the beam search generation process.

The experimental results of the PPP performance comparison of each method at Beams=1, 3, and 5 are shown in Fig. 7, where it can be observed that regardless of the Beams parameter setting of 1, 3, and 5, the chain-of-thinking approach method proposed in this study outperforms all the control baseline methods in terms of the PPP metrics and achieves optimal remediation. In particular, at Beams=3, the chain-of-thinking approach method showed the greatest improvement over other baseline methods, with 8.98%, 24.77%, and 37.94% improvement over VulRepair, VRepair, and TFix based on pre-trained models, respectively, and 18.07% improvement over CURE with a decoder-only model. These data fully demonstrate the excellent performance of the chain-of-thinking approach on the vulnerability remediation task.

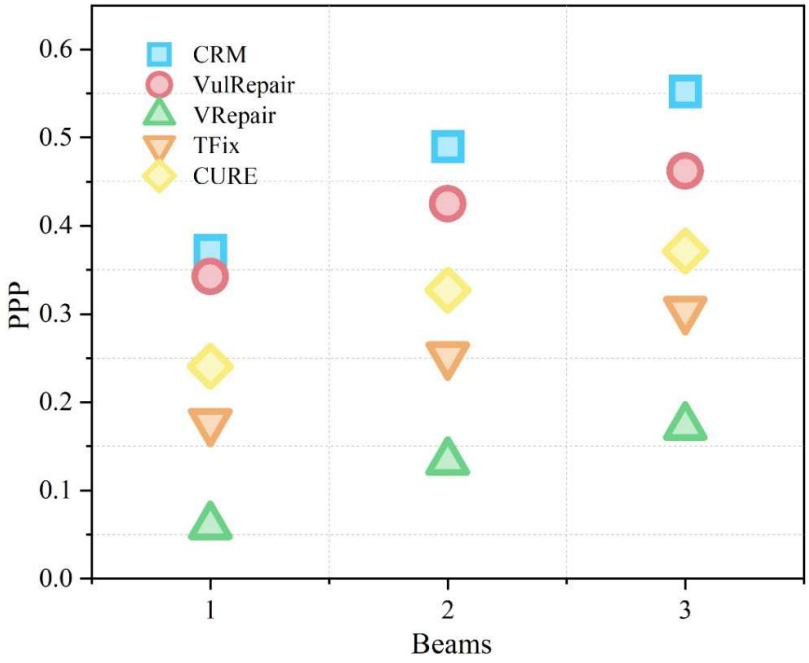

Fig. 7. Comparison results of the PPP performance in different beams

In addition, this study explores the performance of the chain-of-thinking approach over a wider range of Beams values. The PPP and time performance of the modeled thought chain approach in this paper under different Beams is shown in Fig. 8. From the figure, it can be found that the PPP metrics and the average repair time of the chain-of-thinking method show an upward trend as the Beams value increases. When the Beams value is set to 15, the chain-of-thinking approach achieves a PPP of 61.52%, which means that at least 369 vulnerability functions are correctly repaired out of a total of 600 test samples. However, as the Beams value continues to increase, while the increase in the PPP metric tends to moderate, the average repair time shows a linear increase. For example, when the Beams value increases from 15 to 25, the PPP metric only improves by 0.927%, but the fix time increases almost 1.57 times.

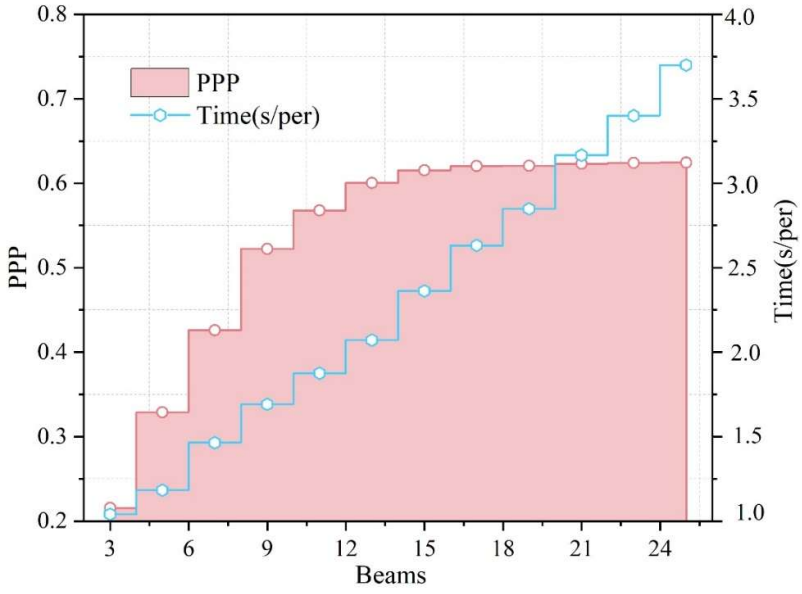

Fig. 8. The PPP and timeliness of this model under different beams

In view of this, this study adopts a weighted summation method that aims to find the best compromise between the PPP indicator and the average restoration time, and the relevant formulas are shown below:

$$Score = w_p \times \overline{P} + w_t \times \overline{T} \tag{26}$$

$$\overline{P} = \frac{p_i - p_{\min}}{p_{\max} - p_{\min}}, i \in [1, 20] \tag{27}$$

$$\overline{T} = 1 - \frac{t_i - t_{\min}}{t_{\max} - t_{\min}}, i \in [1, 20] \tag{28}$$

where Score denotes the normalized score, w_p and w_t denote the weights of the PPP indicator and the average restoration time, respectively, and the sum of the two is 1; $\overline{P}$ and $\overline{T}$ denote the normalized PPP and the average restoration time, respectively; and p_i, $p_{\max}$ and $p_{\min}$ represent the current PPP value, the maximum and minimum values of PPP, respectively. t_i, $t_{\max}$ and $t_{\min}$ represent the current time, the longest time and the shortest time, respectively. Ultimately, in order to maximize the PPP indicator and minimize the average restoration time, the weight ratio of $w_p : w_t = 4:1$ was selected for weighted summation in this study to obtain a result reflecting the PPP indicator and the average restoration time, and the result of the tradeoff between the PPP and the average restoration time is shown in Figure 9. The normalized scores were maximized at a Beams value of 15. Therefore, this study determines that the optimal repair Beams value for the chain of thought approach is 15, when the PPP metric for repairing each vulnerability function is 61.52% and the average time is 3.168 seconds.

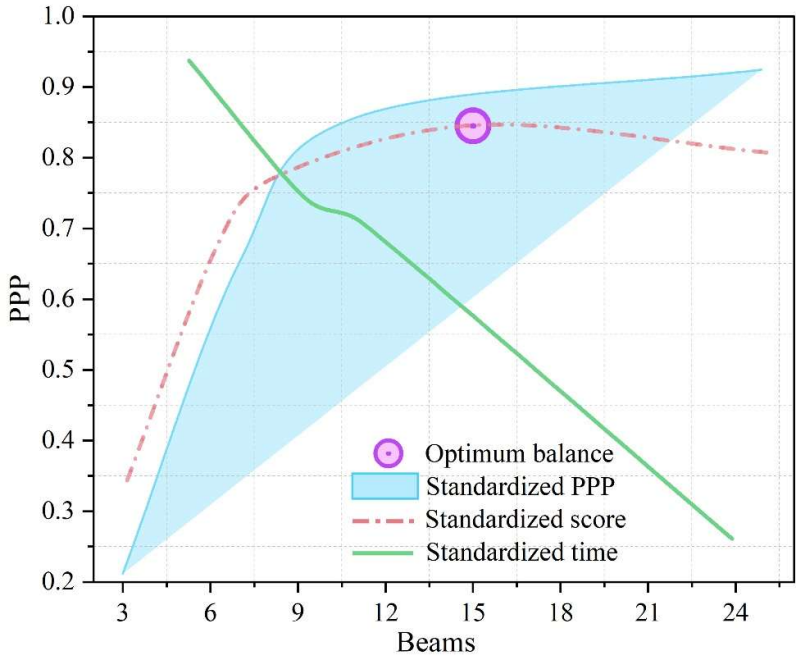

Fig. 9. The compromise between the PPP and the time of repair

4. Conclusion

In order to improve the efficiency of vulnerability mining and the quality of remediation, this paper proposes a vulnerability mining system based on source code semantic graph and a vulnerability remediation method based on chain of thought to provide technical support for the security management of software supply chain.

The main conclusions are as follows:

1) The method in this paper can effectively realize the accurate mining of web service software vulnerabilities, and the model consumes the shortest delay, the smallest fluctuation interval, the strongest stability, and the accuracy of the model for web service software vulnerability mining is above 95%.
2) The model in this paper has obvious advantages in the detection of network service software vulnerabilities, and the precision rate, recall rate, accuracy rate and F1_score value of the model always remain above 80% under different network structures. And after the experiments, the measure of its integration centrality and the optimal number of iteration rounds (60) are determined.
3) The results of the vulnerability repair experiments demonstrate the effectiveness of the chain-of-thinking-based repair method. By evaluating the comprehensive performance of this paper's method under different Beams values, the optimal repair Beams value of 15 is obtained, at which time the PPP index for repairing each vulnerability function is 61.52% and the average time is 3.168 seconds.

Funding

This research was supported by the Science and Technology Major Projects of CNOOC Energy Technology & Services Limited: “Research and Application of Software Supply Chain Security Testing Techniques” (HYFZ-ZX-XK-2022-02).

References

[1] Chong, Y., & Nizam, I. (2018). The impact of accounting software on business performance. International Journal of Information System and Engineering, 6(1), 1-25.

[2] Alt, R., Leimeister, J. M., Priemuth, T., Sachse, S., Urbach, N., & Wunderlich, N. (2020). Software-defined business: Implications for IT management. Business & Information Systems Engineering, 62, 609-621.

[3] Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabber’s knife collection: A review of open source software supply chain attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17 (pp. 23-43). Springer International Publishing.

[4] Enck, W., & Williams, L. (2022). Top five challenges in software supply chain security: Observations from 30 industry and government organizations. IEEE Security & Privacy, 20(2), 96-100.

[5] Boiko, A., Shendryk, V., & Boiko, O. (2019). Information systems for supply chain management: uncertainties, risks and cyber security. Procedia computer science, 149, 65-70.

[6] Singi, K., RP, J. C. B., Podder, S., & Burden, A. P. (2019, November). Trusted software supply chain. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1212-1213). IEEE.

[7] Martínez, J., & Durán, J. M. (2021). Software supply chain attacks, a threat to global cybersecurity: SolarWinds’ case study. International Journal of Safety and Security Engineering, 11(5), 537-545.

[8] Hammi, B., & Zeadally, S. (2023). Software supply-chain security: Issues and countermeasures. Computer, 56(7), 54-66.

[9] Andreoli, A., Lounis, A., Debbabi, M., & Hanna, A. (2023). On the prevalence of software supply chain attacks: empirical study and investigative framework. Forensic Science International: Digital Investigation, 44, 301508.

[10] Lamb, C., & Zacchiroli, S. (2021). Reproducible builds: Increasing the integrity of software supply chains. IEEE Software, 39(2), 62-70.

[11] Shen, Y., Gao, X., Sun, H., & Guo, Y. (2025). Understanding vulnerabilities in software supply chains.

Empirical Software Engineering, 30(1), 1-38.

[12] Ruel, S., Shaaban, S., & Ducros, M. (2019). Supply chain vulnerability: contributions from an edifying case study. Journal of Enterprise Information Management, 32(2), 214-232.

[13] Shourya, R., Kumagai, Y., Yamazaki, H., & Nakakoji, H. (2023). Proposal of Vulnerability Assessment Tool for Software Supply Chain Security. Journal of Information Processing, 31, 842-850.

[14] Islam, S., Abba, A., Ismail, U., Mouratidis, H., & Papastergiou, S. (2022). Vulnerability prediction for secure healthcare supply chain service delivery. Integrated Computer-Aided Engineering, 29(4), 389-409.

[15] Sen, R., Choobineh, J., & Kumar, S. (2020). Determinants of software vulnerability disclosure timing. Production and Operations Management, 29(11), 2532-2552.

[16] Khedr, A. M. (2024). Enhancing supply chain management with deep learning and machine learning techniques: A review. Journal of Open Innovation: Technology, Market, and Complexity, 100379.

[17] Awan, U., Kanwal, N., Alawi, S., Huiskonen, J., & Dahanayake, A. (2021). Artificial intelligence for supply chain success in the era of data analytics. The fourth industrial revolution: Implementation of artificial intelligence for growing business success, 3-21.

[18] Deng, L., & Yu, D. (2014). Deep learning: methods and applications. Foundations and trends® in signal processing, 7(3–4), 197-387.

[19] Mu, R., & Zeng, X. (2019). A review of deep learning research. KSII Transactions on Internet and Information Systems (TIIS), 13(4), 1738-1764.

[20] Dhotre, D., Chandre, P. R., Khandare, A., Patil, M., & Gawande, G. S. (2023). The rise of crypto malware: leveraging machine learning techniques to understand the evolution, impact, and detection of cryptocurrency-related threats. International Journal on Recent and Innovation Trends in Computing and Communication, 11(7), 215-222.

[21] Arpteg, A., Brinne, B., Crnkovic-Friis, L., & Bosch, J. (2018, August). Software engineering challenges of deep learning. In 2018 44th euromicro conference on software engineering and advanced applications (SEAA) (pp. 50-59). IEEE.

[22] Restuccia, F., D'oro, S., & Melodia, T. (2018). Securing the internet of things in the age of machine learning and software-defined networking. IEEE Internet of Things Journal, 5(6), 4829-4842.

[23] Ewelina Mitera Kiełbasa & Krzysztof Zima. (2024). Automated Classification of Exchange Information Requirements for Construction Projects Using Word2Vec and SVM. Infrastructures,9(11),194-194.

[24] Zuo Fei & Rhee Junghwan. (2024). Vulnerability discovery based on source code patch commit mining: a systematic literature review. International Journal of Information Security,23(2),1513-1526.

[25] Palacios Chavarro Sara,Nespoli Pantaleone,DíazLópez Daniel & Niño Roa Yury. (2022). On the Way to Automatic Exploitation of Vulnerabilities and Validation of Systems Security through Security Chaos Engineering. Big Data and Cognitive Computing,7(1),1-1.

[26] Jian Luo,Zechao Chen,Wenxiong Chen,Huali Lu & Feng Lyu. (2024). A study on the application of the T5 large language model in encrypted traffic classification. Peer-to-Peer Networking and Applications,18(1),1-13.