



Article

Deep Forest and Pruned Syntax Tree-Based Classification Method for Java Code Vulnerability

Jiamao Ding ^{1,2}, Weikang Fu ^{1,2}  and Lianyin Jia ^{1,2,*} 

¹ Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650500, China

² Artificial Intelligence Key Laboratory of Yunnan Province, Kunming University of Science and Technology, Kunming 650500, China

* Correspondence: lianyinjia@kust.edu.cn

Abstract: The rapid development of J2EE (Java 2 Platform Enterprise Edition) has brought unprecedented severe challenges to vulnerability mining. The current abstract syntax tree-based source code vulnerability classification method does not eliminate irrelevant nodes when processing the abstract syntax tree, resulting in a long training time and overfitting problems. Another problem is that different code structures will be translated to the same sequence of tree nodes when processing abstract syntax trees using depth-first traversal, so in this process, the depth-first algorithm will lead to the loss of semantic structure information which will reduce the accuracy of the model. Aiming at these two problems, we propose a deep forest and pruned syntax tree-based classification method (PSTDF) for Java code vulnerability. First, the breadth-first traversal of the abstract syntax tree obtains the sequence of statement trees, next, pruning statement trees removes irrelevant nodes, then we use a depth-first based encoder to obtain the vector, and finally, we use deep forest as the classifier to get classification results. Experiments on publicly accessible vulnerability datasets show that PSTDF can reduce the loss of semantic structure information and effectively remove the impact of redundant information.

Keywords: vulnerability classification; abstract syntax tree; code representation; deep forest

MSC: 68T10



Citation: Ding, J.; Fu, W.; Jia, L. Deep Forest and Pruned Syntax Tree-Based Classification Method for Java Code Vulnerability. *Mathematics* **2023**, *11*, 461. <https://doi.org/10.3390/math11020461>

Academic Editors: Heui Seok Lim, Sanghyuk Lee, Yeongwook Yang and Imatitkua Aiyanyo

Received: 24 December 2022

Revised: 10 January 2023

Accepted: 12 January 2023

Published: 15 January 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The rapid development of J2EE (Java 2 Platform Enterprise Edition) has brought unprecedented severe challenges to Java source code vulnerability mining. According to the data released by the vulnerability knowledge base website CVE details [1], information security companies and security researchers submitted 24,172 vulnerabilities to the Common Vulnerabilities & Exposures (CVE) in 2022, an increase of 4001 from 20,171 in 2021. In addition to the increase in the number, the forms of software vulnerabilities also show complexity and diversity, and the threats to the normal and safe operation of computer systems are increasing day by day.

Since the classifier cannot understand the source code in text form, it needs to convert a source code file into a vector that contains the syntax and semantic structure information of the code. This process is called code representation. The code representation method based on the abstract syntax tree (AST) is better than the code representation method based on code metrics [2] and sequences of tokens [3]. It comprehensively considers the code semantics and structural information and thus is more comprehensive. In the existing studies based on abstract syntax tree representation, most studies do not eliminate irrelevant nodes such as annotations and package references, these irrelevant nodes will lead to long training time and overfitting. Another problem is that most studies use the depth-first algorithm to traverse the whole tree to collect node information when processing

AST. This algorithm may convert two different ASTs into the same sequence of tree nodes, resulting in some AST structural information loss and a decrease in accuracy.

The novelty and contribution of our research are as follows:

1. The breadth-first algorithm is used to solve the problem of semantic structure information loss when processing abstract syntax trees. We are the first to propose an algorithm that uses breadth-first to process abstract syntax trees to obtain expression subtrees, which solves the problem of semantic structure information loss caused by code structures being translated to the same sequence of tree nodes when using depth-first to process abstract syntax trees.
2. We proposed a statement tree pruning algorithm to solve the problem of information redundancy caused by irrelevant information (such as package references and comments) in source code. The irrelevant nodes in the abstract syntax tree are not helpful for classification but will lead to long training time and overfitting. To solve this problem, we propose pruning the statement tree obtained from the transformation of the abstract syntax tree to solve the problem of redundant information.

The structure of this paper is as follows: Section 1 introduces the research background, novelty, and contribution of this paper; Section 2 introduces the recent works and progress; Section 3 introduces the composition of the PSTDF (pruned statement tree-based deep forest) in detail; Section 4 contains five experiments. In this part, we have carried out validation experiments and comparative experiments on different datasets and then analyzed each experiment to verify that PSTDF can reduce the loss of semantic information and remove the negative impact of irrelevant information. Section 5 discusses the advantages and limitations of PSTDF; Section 6 summarizes the work of this paper and looks forward to the work of the next step.

2. Related Work

The vulnerability feature extraction and classification method in Java source code is the main research work of this paper. First of all, we need to extract the features of each type of vulnerability in the source code file. After getting the features, we convert them into vector form which can be accepted by the machine. At the same time, the converted vectors can represent the essence of various vulnerability codes as much as possible. Finally, the source codes are classified based on the extracted features. Therefore, our work mainly includes two aspects: representation of vulnerability source code and classifier testing.

The existing abstract syntax tree-based code representation methods can be divided into two types according to different feature extraction methods, one is path information representation. Ref. [4] proposes to decompose the tree into paths, and the path information obtained from the decomposition is used as the input of the transformer for code prediction; Ref. [5] proposes the PATA model, which distinguishes multiple occurrences of the same variable based on execution path information for stain analysis; Ref. [6] designs a model for training deep learning classifiers to predict the correctness of patches by traversing the AST path in depth-first.

The other is node information representation. For example, Ref. [7] proposes an AST-based neural network (ASTNN) to divide the AST into statement trees and perform depth-first traversal on the statement trees to obtain statement sequences and convert them into 128-dimensional vectors, and then train a bidirectional RNN (recurrent neural network) model for code clone detection and function classification; Ref. [8] divides the AST into statement trees and performs depth-first traversal on the statement trees to obtain sequences of a statement, then uses Word2Vec [9] to translate each statement in the sequence into a vector, and finally uses the obtained vector as the input of the BiLSTM (bi-directional long short-term memory) [10] model. Ref. [11] proposes a transformer-based source code classification method on the basis of [7] to avoid gradient disappearance and long-distance dependence. Ref. [12] uses the depth-first algorithm to process the AST to obtain the sequence of the AST nodes, and then uses the continuous bag-of-words model to generate word vectors, which are sent to a model built on GAN (generative adversarial network) [13]

for training, and finally uses this model for software defect prediction. The authors of [14] propose the TreeCaps model on the basis of Mou et al. [15], which uses the capsule network with the tree-based convolutional neural network. TreeCaps achieves higher learning accuracy than existing graph-based neural networks.

In terms of classifier selection, some users may not want their source code to be uploaded to the public server without supervision in actual vulnerability detection. To solve this problem, Ref. [16] designs a sparse autoencoder (SAE) based federated learning framework consisting of two independent classifiers to accurately compute classification results while preserving privacy; Ref. [17] proposes a model based on federated learning to protect privacy. This model introduces the concept of reputation as a measure to achieve effective reputation management of public servers under the condition of undeniable and tamper-proof. In addition, the authors of [18,19] propose using a semantic-complete graph (SMGA) for classification. In this model, a graph-embedded semantic completion module (GSC) is designed to complete mismatched semantics by generating hallucination graph nodes in missing categories to overcome significant intra-class variance and domain mismatched semantics in training batches, which led to the problem of suboptimal adaptation.

The above representation methods based on the AST have achieved good results in the fields of vulnerability mining and defect prediction. However, there are still two problems: most studies [4–8] use depth-first algorithms to deal with abstract syntax trees, and different code structures will be translated to the same sequence of tree nodes when processing abstract syntax trees using depth-first traversal, this leads to a loss of semantic structure information which will reduce the accuracy of the model. Another problem is that some studies [7,8,11] do not exclude redundant information brought by irrelevant nodes when dealing with abstract syntax trees, which will lead to longer training time and overfitting. Inspired by [7], we propose a vulnerability classification method based on improved statement tree representation. This method first performs a breadth-first traversal on the AST according to the concept of a statement tree to convert it into a series of statement trees and combines these statement trees into a sequence according to the traversal order. Then, to reduce information redundancy, we prune each statement tree in the sequence to remove irrelevant nodes such as annotates and package references. Next, depth-first traversal of every statement tree in the sequence is performed to obtain a sequence of statements and encode each statement in sequence into a vector by using Word2Vec. Finally, we use the deep forest for classification.

Our experiments prove that the method proposed by us can decrease the loss of semantic structure information effectively and eliminate the negative impact of irrelevant redundant information in AST.

3. Model Building

The pruned statement tree-based deep forest (PSTDF) proposed by us is mainly divided into three stages. Stage one is data preprocessing; stage two is vulnerability feature extraction; stage three is vulnerability code classification. As shown in Figure 1, the AST is traversed by using the breadth-first algorithm to obtain the sequence of statement trees, and then the node pruning is performed on each statement tree in the sequence. Next, each statement tree is converted into the vector through the encoding layer and pooling layer. Finally, we use the deep forest to classify the code.

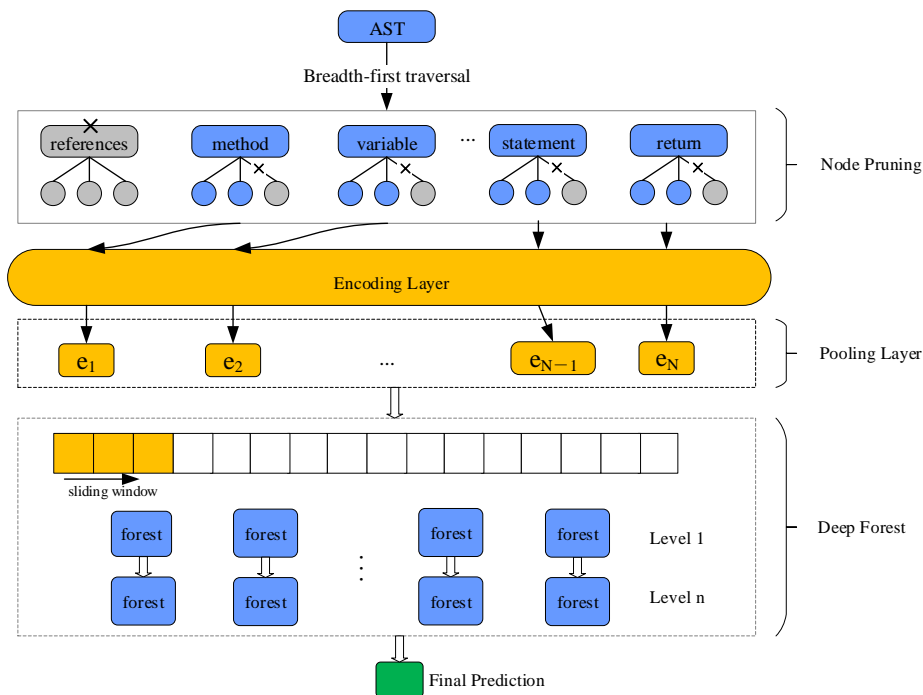


Figure 1. The architecture of the PSTDF.

3.1. Parse Source Code into AST

Since the classifier cannot understand the source code in the form of text, we need to convert a code file into a vector containing the semantic structure of the code, and then input it into the classifier for training. An abstract syntax tree (AST) is an intermediate representation in the code compilation process. The AST node stores the grammatical structure information of the code [20]. The AST can well reflect source code semantic structure and syn-tax information. In the experiments in this paper, we use javalang to complete the task of parsing source code into an AST. Javalang is a Python library that can be installed directly using pip, and it can parse the entire Java source code file or code fragment into the corresponding AST. Figure 2a,b show a piece of Java code and converted AST.

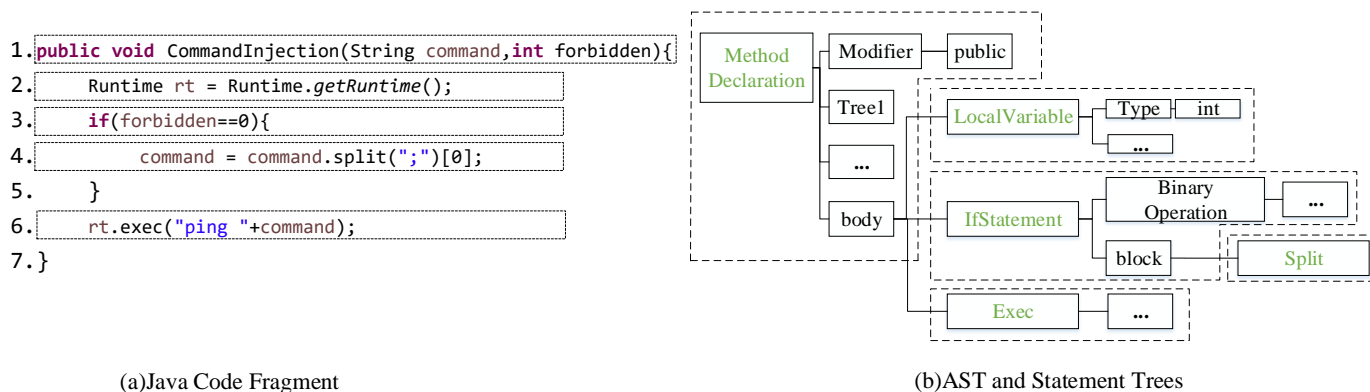


Figure 2. Java code converted to statement trees.

3.2. Obtain the Sequence of Statement Trees

After the AST is obtained, the AST needs to be traversed breadth-first to obtain the sequence of statement trees for further processing.

Formally, given an AST's statement node set S , each statement in the code corresponds to an element $s \in S$. For nested statements such as those in Figure 2b, we define a set of independent nodes: $P = \{block, body\}$. $body \in P$ represents the method declaration in the scope, and $block$ is used to split the declaration and scope of nested statements (for example, *while* statement, *try – catch* statement, and so on). Furthermore, we define a set $D(s)$. This set consists of all of the children of $s \in S$. For any $d \in D(s)$, if d can reach s through a node in P , then d is included in the statement represented by the node s , and we call d a sub-statement node of s .

According to the above definition, a statement tree is a tree whose root is the statement node $s \in S$ and consists of sub-statement nodes of s (not including the statement nodes in the set P). It has been demonstrated in Ref. [7] that the granularity of the statement tree can achieve a good balance between the depth, size, and diversity of the semantic structure of the AST.

As shown in Figure 3, according to the definition of the statement tree, the depth-first algorithm can be used to traverse the AST to obtain a sequence of statement trees.

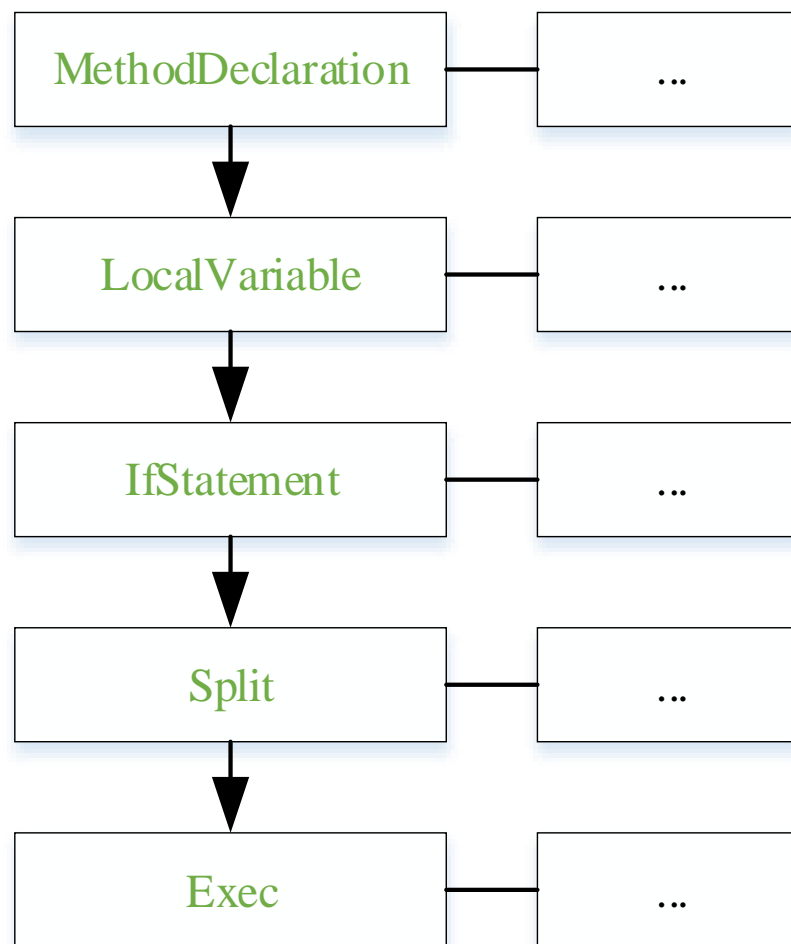


Figure 3. Sequence of statement trees. This sequence was obtained from the AST in Figure 2b.

However, the depth-first algorithm used in [7] and most studies based on AST when dealing with AST will have the problem of tree structure information loss. We take the two methods in Figure 4a as an example (for simplicity, only the root nodes in the statement tree were drawn to illustrate the problem), the java method “danger” is a method with command injection vulnerability (CWE78), the method “safe” is a method without CWE78.

```

1. public void danger(String command,int forbidden) {
2.     Runtime rt = Runtime.getRuntime();
3.     if(forbidden==0) {
4.         command = command.split(";")[0];
5.     }
6.     rt.exec("ping "+command);
7. }
    
```

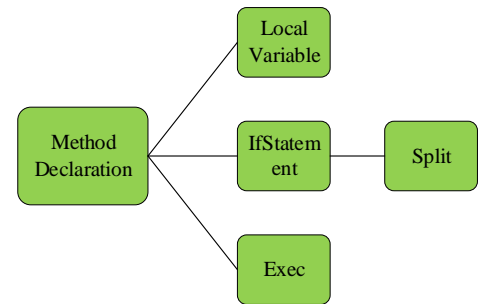
Danger method

```

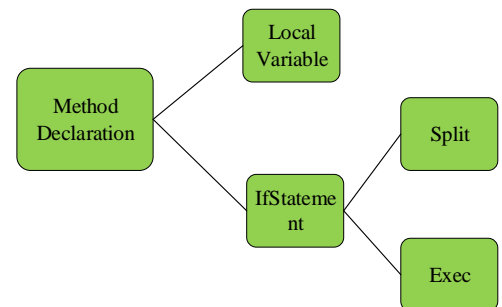
1. public void safe(String command,int forbidden){
2.     Runtime rt = Runtime.getRuntime();
3.     if(forbidden==0) {
4.         command = command.split(";")[0];
5.         rt.exec("ping "+command);
6.     }
7. }
    
```

Safe method

(a)Java Code Fragments



AST1

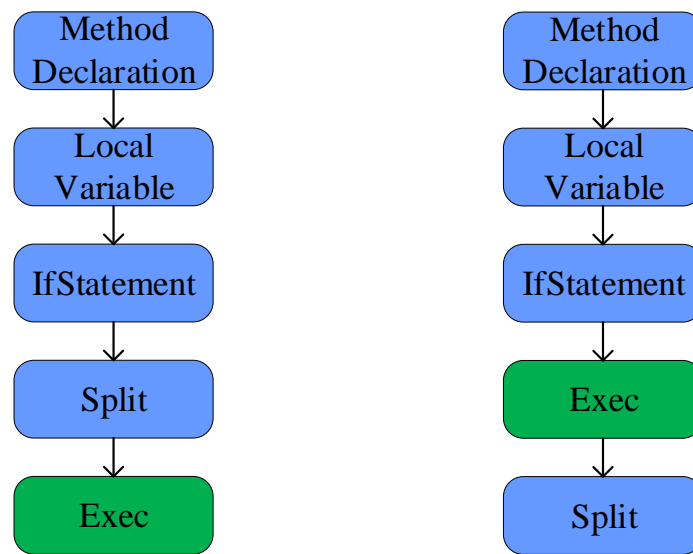


AST2

(b)ASTs

Figure 4. Comparison of breadth-first traversal and depth-first traversal. In this Figure, both Java methods are converted to AST1 by depth-first traversal, and the safe method without vulnerability is converted to AST2 by breadth-first traversal.

In the danger method, if we pass in the parameter “forbidden = 1”, the security processing will be bypassed and any system command we want will be executed. The safe method does not have this vulnerability, no matter what parameters are input, no unexpected command will be executed. The ASTs parsed by the two methods are shown in Figure 4b. When the depth-first algorithm is used to traverse the two ASTs, both trees will be converted into the same sequence of statement trees in Figure 5a. These two methods have only slight differences in code structure, while the depth-first traversal-based method in the literature cannot capture such differences. We propose to use breadth-first traversal to solve this problem. When breadth-first traverses two ASTs, the safe method will be converted into the sequence of statement trees shown in Figure 5a, and the danger method will be converted into the sequence shown in Figure 5b. It can be seen that different semantic information of the two ASTs has been preserved and we avoid the loss of this part of the semantic information.



(a) Statement Tree Sequence 1

(b) Statement Tree Sequence 2

Figure 5. Two kinds of Statement Tree Sequence and differences. Sequence 1 is obtained from AST1 in Figure 4b by breadth-first traversal, and Sequence 2 is obtained from AST2 in Figure 4b by breadth-first traversal.

The depth-first traversal-based AST parsing algorithm proposed in [7] is shown in Algorithm 1, and our improved algorithm is shown in Algorithm 2.

Algorithm 1 Original AST parsing algorithm.

Input: AST root node

Output: sequence of statement trees

```

1: Function dfs(root,sequence){
2:   children = root.children; //get children of root
3:   sequence.add(root);
4:   for child:children do //recursive traversal
5:     dfs(child,sequence);
6:   end for
7:   if root ∉ S ∪ P then //base case
8:     return;
9:   end if
10: }
  
```

As shown in Algorithm 2, given an AST T , breadth-first traversal is performed starting from the root node, and first, it is judged whether the current node is a node in the statement set S if it is a node in S , put it into the sequence of statement trees, if not, it is further judged whether the current node belongs to the nested node set $P = \{block, body\}$. If a current node is in the set P , we put it into the sequence of statement trees, if not, it continues to traverse until the end of the loop, and the algorithm outputs the sequence of statement trees.

Algorithm 2 Improved AST parsing algorithm.**Input:** AST root node**Output:** sequence of statement trees

```

1: res,queue = []; //init
2: queue.add(root); //add root into queue
3: while queue do //breadth-first traversal
4:   node = queue.pop(); //head element out
5:   if node ∈ S ∪ P then
6:     res.add(node); //determine whether the current node is a statement node
7:   end if
8:   for child:node.children do
9:     queue.add(child);
10:  end for
11: end while
12: return res;

```

3.3. Irrelevant Node Pruning

The method proposed in [7] was originally designed for function-level code fragments. In the original model, all statement trees and their nodes are encoded, resulting in some irrelevant nodes such as package reference nodes and annotation nodes being encoded as features, and these nodes are encoded as features that will increase model training time and even lead to overfitting. To remove redundant information, the statement tree is pruned before the encoding process, irrelevant nodes such as annotations and package references will be removed, and only necessary semantic information is retained for encoding. We define a set of the irrelevant node set $U = \{import, annotate\}$.

As shown in the node pruning layer in Figure 1, the sequence of statement trees obtained by Algorithm 2 is traversed once, and firstly, it is judged whether the root node of each statement tree belongs to U , if the root node belongs to U , the statement tree will be discarded. If it does not belong to U , the child of the root node is traversed once, if there is a child node belonging to the set U , this child will be discarded too. Finally, we get the pruned sequence of statement trees $[T_1, T_2, \dots, T_{N-1}, T_N]$. The above process is shown in Algorithm 3.

Algorithm 3 Node pruning algorithm.**Input:** sequence of statement trees**Output:** pruned sequence of statement trees

```

1: queue = []; //init output sequence
2: for root : res do
3:   if root ∉ U then //discard useless statement tree
4:     for child : root.children do
5:       if child ∈ U then
6:         root.remove(child); //discard useless tree node
7:       end if
8:     end for
9:     queue.add(root);
10:  end if
11: end for
12: return queue;

```

3.4. Encode Sequence of Statement Treess as Vector

When encoding, we use the sequence of statement trees $[T_1, T_2, \dots, T_{N-1}, T_N]$ obtained from Algorithm 3 as input. The encoder will encode each statement tree T_N in the sequence. Taking the statement tree in Figure 6 as an example, first, depth-first traversal on the statement tree is performed, and we obtain the node label set as corpus and then train

Word2Vec by using this corpus to obtain the node index list. Then, the second depth-first traversal of the statement tree is performed to obtain the node set $nodes$, and each statement tree node in $nodes$ is converted to d -dimensional vector v_n through the Word2Vec model obtained during the first traversal.

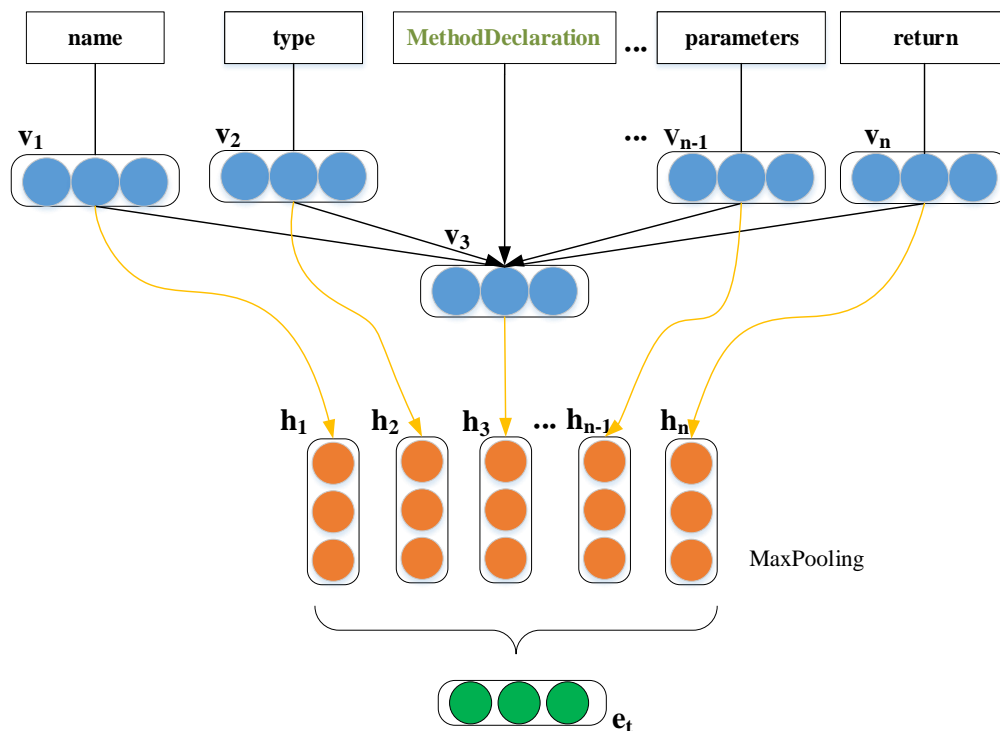


Figure 6. The structure of statement tree encoder. This encoder uses depth-first traversal to encode a statement tree into a vector.

Concretely, given a statement tree node $n \in nodes$ which has no leaf, first, we use Equation (1) to perform word embedding on node n to obtain the vector representation v_n of n . In Equation (1), x_n is one-hot representation of node n , $\mathbf{W}_e^T \in \mathbb{R}^{|V| \times d}$ is a weight matrix, d is the word embedding dimension, V is the vocabulary size, and \top is matrix transpose operation.

$$v_n = \mathbf{W}_e^T x_n \tag{1}$$

Then, we use Equation (2) to recursively update the vector v_n so that it contains child nodes and hierarchical structure information to obtain the vector list $[h_1, h_2, \dots, h_{n-1}, h_n]$. In Equation (2), $\mathbf{W}_n^T \in \mathbb{R}^{k \times d}$ is the k -dimension encoding weight matrix, \top is matrix transpose operation, d is the word embedding dimension, C is the number of child nodes of node n , b_n is a bias term, h_i is updated vector of the child nodes of node n , and \tanh is the activation function.

$$h = \tanh(\mathbf{W}_n^T \in \mathbb{R}^{k \times d} + \sum_{i \in [1, C]} h_i + b_n) \tag{2}$$

Because the number of $n \in nodes$ in each statement tree is variable, it is necessary to perform a transposition operation on the updated vector list $[h_1, h_2, \dots, h_{n-1}, h_n]$. As shown in Equation (3), n is the number of statement tree nodes which has no leaf, and d is the word vector dimension.

$$\begin{bmatrix} h_{(1,1)} & h_{(1,2)} & \cdots & h_{(1,d)} \\ h_{(2,1)} & h_{(2,2)} & \cdots & h_{(2,d)} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(n,1)} & h_{(n,2)} & \cdots & h_{(n,d)} \end{bmatrix} \xrightarrow{\text{transpose}} \begin{bmatrix} h_{(1,1)} & h_{(2,1)} & \cdots & h_{(n,1)} \\ h_{(1,2)} & h_{(2,2)} & \cdots & h_{(n,2)} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(1,d)} & h_{(2,d)} & \cdots & h_{(n,d)} \end{bmatrix} \tag{3}$$

Then, as shown in Equation (4), the maximum pooling operation is performed on the transposed vector list line by line to obtain the vector representation e_t of the statement tree T .

$$e_t = [\max(h_{i,1}), \max(h_{i,2}), \dots, \max(h_{i,n})], i \in [1, d] \quad (4)$$

We encode all statement trees in the sequence one by one to obtain the final vector representation $e_i = [e_1, e_2, \dots, e_t, \dots, e_N], i \in [1, N]$ of an AST, where N is the length of the sequence of statement trees.

Finally, as shown in Figure 1, e_i is converted into a fixed-length vector through the pooling layer constructed using Equations (3) and (4) and is used as an input of the classifier.

3.5. Vulnerability Classification

To further reduce the model's time cost under the premise of ensuring accuracy, we choose deep forest as the classifier. Deep forest is an integrated forest structure proposed in [21], which can work well with small samples and has low computational overhead.

Since the number of samples in vulnerability datasets is usually not very large, we use multi-grained scanning to increase the diversity of samples. As shown in Figure 1, the final vector representation $e_i = [e_1, e_2, \dots, e_t, \dots, e_N]$ of an AST enters the multi-grained scanning stage, and we use a window of length 1 to scan by step size 1 on the samples of the training set. Next, we use the features in each window as the input of this stage (multi-grained scanning) and output the probability vector of the features in each window. Finally, we concatenate all the probability vectors as the input of the next stage (cascade forest).

The cascade forest has several layers, as shown in Figure 1, and each layer consists of several random forests and completely random forests. The first layer takes the probability vector output by the multi-grained scanning stage as input. Additionally, the output of the previous cascade layer is used as the input of the next cascade layer. During the whole training process, every time a layer of cascade forest is added, k-fold cross-validation is used to test the generated cascade forest. If the accuracy rate is lower than the accuracy rate of the previous layer, the number of cascade forest layers will not increase. Otherwise, the cascade layer continues to increase.

After the cascade forest stops growing, the cascade forest averages all the probability vectors output by the last layer and outputs the label category with the highest probability as the predictive value of the final vulnerability classification.

4. Experiments

4.1. Experiment Settings and Dataset Description

To verify the feasibility and performance of the PSTDF, four sets of comparative experiments are designed.

Experiment Settings: We use Python3.6 at parsing the source code into AST, statement tree encoding, and deep forest classification. All experiments are run on a laptop equipped with an AMD RYZEN 7 octa-core processor, 64 GB of memory, and a 64-bit Windows 10 operating system. Except for the model using the deep forest and the traditional machine learning model running on the CPU, the rest of the neural network models are all running on a GTX1650 graphics card with 4 GB of video memory.

Dataset Description: The vulnerability data used in this paper comes from the OWASP (Open Web Application Security Project) benchmark vulnerability dataset v1.1 and the SARD (Software Assurance Reference Dataset) JulietJava dataset. The OWASP dataset contains a total of 21,041 operational use cases including 11 types of vulnerabilities such as command injection, weak encryption algorithm, SQL injection, and directory traversal. Each category contains positive samples with vulnerabilities and negative samples without vulnerabilities. The ratio of positive and negative samples in the OWASP dataset is about 1.28:1. The information about the OWASP dataset is shown in Table 1.

Table 1. Basic information about the OWASP dataset.

CWE Number	Vulnerability Type	Positive Samples	Negative Samples	Total
CWE78	cmdi	906	1802	2708
CWE327	crypto	720	720	1440
CWE328	hash	707	714	1421
CWE90	LDAP	215	521	736
CWE22	pathtraver	924	1706	2630
CWE614	securecookie	215	201	416
CWE89	sqli	1232	2297	3529
CWE501	trustbound	220	505	725
CWE330	weakrand	2028	1612	3640
CWE643	XPATH	130	217	347
CWE79	XSS	1909	1540	3449

In addition to the OWASP dataset, we also select 12 CWE-numbered vulnerabilities from the SARD JulietJava dataset to construct a subset containing 15,340 samples. Since each Java file in the SARD dataset contains two kinds of functions: bad (with vulnerabilities) and good (without vulnerabilities), the ratio of positive and negative samples in the subset constructed based on the SARD dataset is 1:1. The detailed information (vulnerability number, vulnerability type, quantity, etc.) of the SARD dataset is shown in Table 2.

Table 2. Basic information about the SARD dataset.

CWE Number	Vulnerability Type	Positive Samples	Negative Samples	Total
CWE78	cmdi	906	1802	2708
CWE327	crypto	720	720	1440
CWE328	hash	707	714	1421
CWE90	LDAP	215	521	736
CWE643	XPATH	264	264	528
CWE614	securecookie	17	17	34
CWE89	sqli	1320	1320	2640
CWE80	XSS	792	792	1584
CWE15	External Control of System	264	264	528
CWE113	HTTP Request Splitting	792	792	1584
CWE129	Improper Validation of Array	1584	1584	3168

4.2. Experiment 1

The first experiment verified the effectiveness of the improved statement tree representation method proposed by us on the OWASP dataset and analyzed its time complexity.

Experiment parameters: In this experiment, the train set and test set were randomly selected from the OWASP dataset at a ratio of 6:4. The code representation uses the sequence of statement tree (ST) proposed in [7] and the sequence of pruned statement tree (PST) proposed by us, respectively. The classifiers all use the GRU in [7]. We use the skip-gram algorithm to train Word2Vec to convert the sequence of statement trees into the word vector and the word embedding size is 128 dimensions. The hidden dimension of the GRU [22] is 100; the hidden dimension of the original statement tree encoder in the original ASTNN [7] is 100 as well. The batch size is 64 and the epoch is 2. When we train the neural network, we use the optimizer AdaMax [23] with a learning rate of 0.002.

The results are shown in Table 3.

We also calculated the confusion matrix of ST, which is shown in Figure 7, and the confusion matrix of PST is shown in Figure 8.

Result analysis: Taking the AST in Figure 2b as an example, we calculate $f(n)$ and analyzed the time complexity of Algorithms 1 and 2. For Algorithm 1, it is calculated to get $f(n) = n^2 + 2$, and the time complexity of Algorithm 1 is $O(n^2)$. For Algorithm 2, it is

calculated to get $f(n) = 2n + 4$, and the time complexity of Algorithm 2 is $O(n)$. Therefore, we can find that the time complexity of Algorithm 2 is less than that of Algorithm 1. In addition, PST can further reduce time consumption during training.

In terms of memory consumption, Algorithm 2 uses a queue-based non-recursive algorithm, which is also better than Algorithm 1 using recursion in terms of memory consumption. This experiment shows that compared with the original method proposed in [7], PST could effectively reduce the loss of semantic structure information (improved the accuracy and other indicators by about 2.4%) and the negative impact of irrelevant redundant information in AST (reduced the time overhead by about 36%).

Table 3. Basic information about the SARD dataset.

Code Representation	Classifier	Accuracy (%)	Recall (%)	F1 (%)	Time Cost (s)
ST	GRU	93.35	93.54	93.33	675.89
PST	GRU	97.59	97.61	97.48	428.31

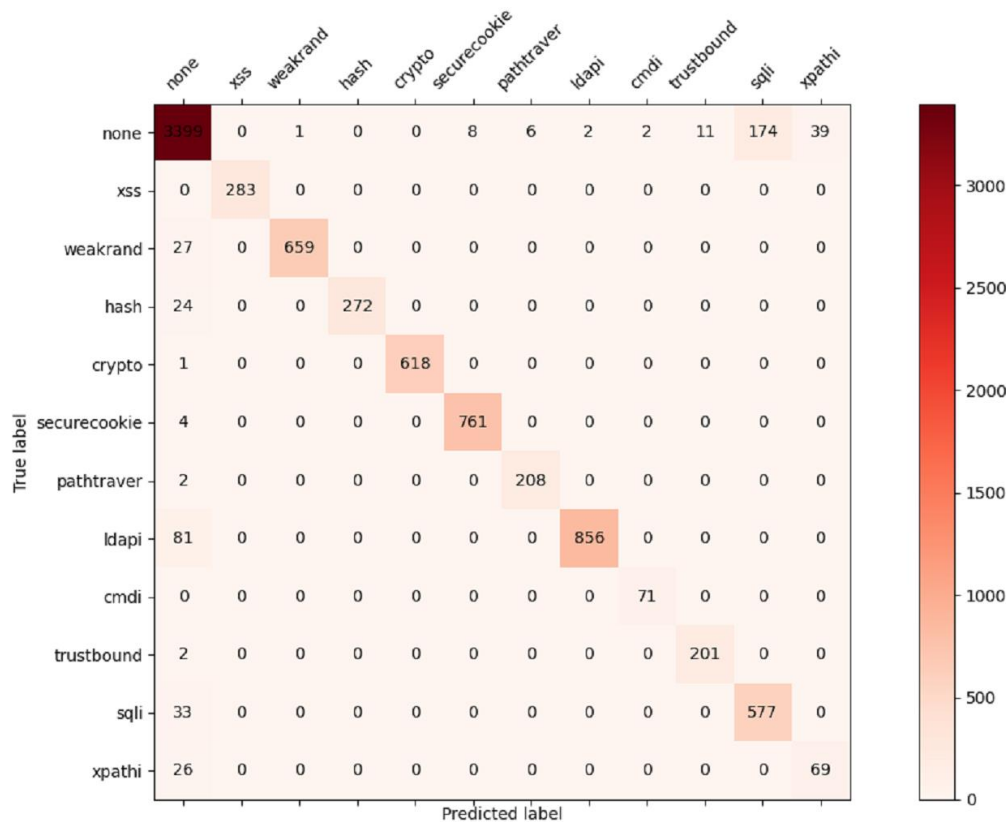


Figure 7. Confusion matrix of ST. This confusion matrix is obtained by using the original statement tree(ST) segmentation algorithm.

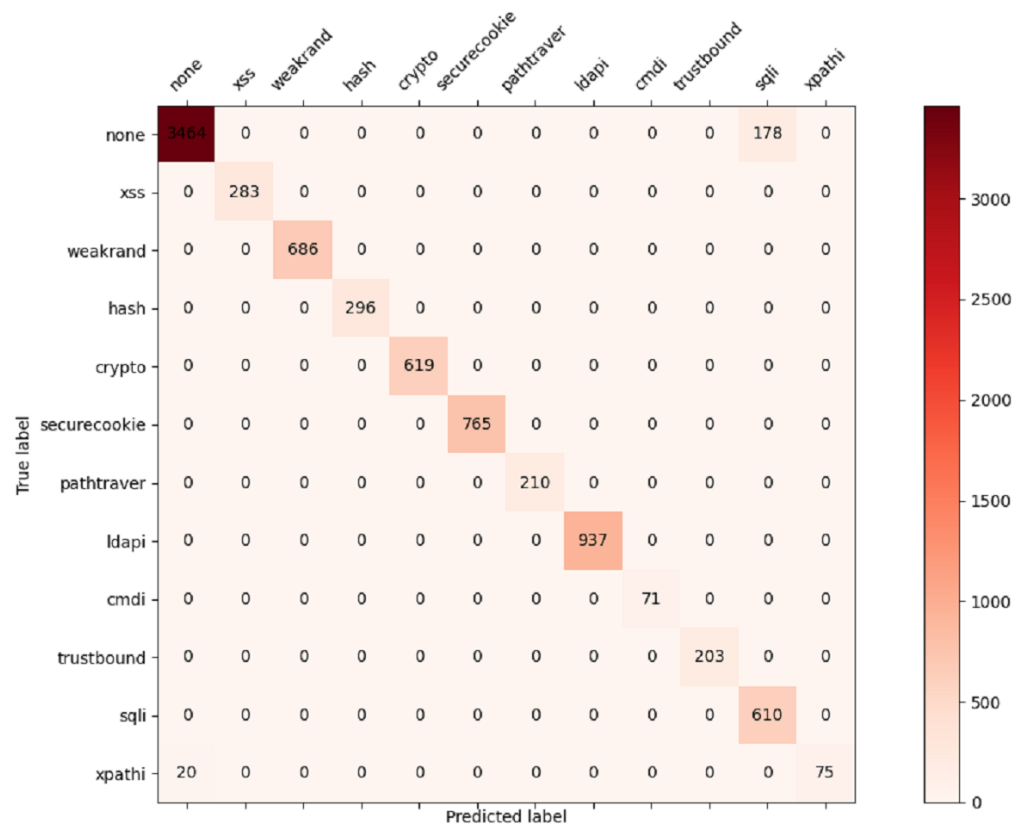


Figure 8. Confusion matrix of PST. This confusion matrix is obtained by using the pruned statement tree(PST) segmentation algorithm.

4.3. Experiment 2

The second experiment used the improved algorithm proposed by us to test the model based on the different classifiers to verify that using deep forest as a classifier is good.

Experiment parameters:

For LSTM (long short-term memory) and BiLSTM, we set the dimensionality of the hidden state as 100. The number of LSTM cells is 128, the epoch is 50, and the batch size is 64. For the CNN (convolutional neural network) [24], we use a neural network containing a convolution layer and a pooling layer set epoch as 500, and the learning rate is 0.1. For SVM (support vector machines), we built an SVM classifier using the liner kernel. For the decision tree, we use the Gini impurity, the maximum depth of the tree is 5. For GRU, the parameters of the GRU used in this experiment are consistent with experiment 1. For deep forest, each forest in the cascade forest has 200 trees, sliding window size and step size are both 1, and the maximum number of cascaded layers is 20.

This experiment is carried out on the OWASP dataset and the results are shown in Table 4.

Table 4. Classifier comparison.

Classifier	Accuracy (%)	Recall (%)	F1 (%)	Time Cost (s)
LSTM	27.20	27.20	24.01	345.19
BiLSTM	32.26	32.26	26.0	322.96
CNN	33.31	33.31	25.14	505.17
SVM	7.28	7.28	5.22	213.34
Decision Tree	13.72	13.72	8.93	216.91
GRU	97.59	97.61	97.48	504.78
DeepForest	99.13	99.13	99.13	376.33

Result analysis: The data in Table 4 shows that although traditional classifiers such as SVM and CNN are fast, they cannot effectively learn the deep features in the code representation. For vulnerability classification, GRU is better than LSTM. GRU has a simpler structure and can solve the problem of long-distance information context dependence, so it is better in accuracy and time consumption. However, deep forest has higher accuracy by generating sub-samples in the multi-grained scanning stage, and it can also achieve a time overhead similar to that of a model running on a GPU when running under a CPU.

4.4. Experiment 3

In the OWASP dataset, we conduct extensive experiments on different models. We compare traditional machine learning models such as SVM-based and other models based on deep neural networks such as MCDF (malicious code classification method based on deep forest) [25], LSTM [26,27], BiLSTM [28], TextCNN [29], and TextGCN [30]. Parameters in each method were shown as follows:

- In TextCNN, LSTM, and BiLSTM, the code is regarded as plain text and converted into sequence of tokens representation as input. For TextCNN, we set the kernel size as 3 and set the number of filters as 100. For LSTM and BiLSTM, we set the dimensionality of the hidden state as 100.
- In SVM, we use the SVM with traditional statical features-based methods such as the TF-IDF (term frequency-inverse document frequency) algorithm, N-gram [31] algorithm, and LDA (linear discriminant analysis) [32] algorithm. These methods extract tokens from Java source code files. For the LDA algorithm, the number of topics is set as 300; for the N-gram algorithm, the number of max features is set as 1000 and the number of grams is set as 2.
- In MCDF, the Java source code was treated as a character stream file, read 8-bit binary numbers into decimal integers, and reshape these integers into fixed-line-width vectors as the deep forest's input vectors.
- We also tested the performance of graph neural networks such as TextGCN. In TextGCN, documents and words in code files are regarded as graphs' nodes; we use the co-occurrence frequency information of words to construct the edge between word nodes, and the document frequency and word frequency are used to construct the edge between different kinds of nodes (word node and document node) to construct a large graph. The graph is then modeled using GCNs (graph convolutional networks), converting the code function classification problem into a node classification problem.

Result analysis: The results are shown in Table 5. From Table 5 it can be seen that traditional models such as SVM did not perform well. Traditional methods such as SVM mainly rely on shallow semantic features and the semantics of tokens to distinguish code functions. However, in this experiment, the OWASP dataset has a similarly large number of annotations and package reference nodes, so token-based methods are not effective in classifying vulnerability source code. For other deep neural network models, BiLSTM and TextCNN are better than the above token-based methods because this type of recurrent neural network can capture more local features in Java code files. In TextGCN, the graph-based approach performs passably well but it uses node numerical ID to represent the nodes of the graph, this numerical ID-based method misses lexical knowledge. Additionally, many graph-based works such as TextGCN only focus on the explicit dependency information on a high level of abstraction [33]. In all models, our model achieves faster speed and the best accuracy. Because PSTDF performs feature capture on the statement tree that is much smaller than the original AST according to the method proposed in the literature [7]. It achieves good results in both class-level files and code fragments. Additionally, the improved algorithm used by PSTDF can save a lot of memory space to increase the calculation speed.

Table 5. Comparison on the OWASP dataset.

Method	Accuracy (%)	Recall (%)	F1 (%)
SVM+TD-IDF	44.16	44.16	27.06
SVM+N-gram	49.93	49.93	33.26
SVM+LDA	43.91	43.91	26.80
LSTM	43.81	8.33	5.08
BiLSTM	89.26	89.26	89.90
MCDF	91.97	85.83	86.59
TextCNN	86.36	86.36	86.30
ASTNN	93.35	93.54	93.33
TextGCN	90.60	90.04	89.96
PSTDF	99.13	99.13	99.13

4.5. Experiment 4

The fourth experiment uses the SARD dataset to test the generalization ability of PSTDF.

Experiment parameters: In this experiment, all models' parameters were consistent with those in Experiment 3.

Result analysis: The results are shown in Table 6. From Table 6 it can be known that PSTDF also has good generalization ability on different datasets.

Table 6. Generalization experiment.

Method	Accuracy (%)	Recall (%)	F1 (%)
SVM+TD-IDF	49.40	49.40	33.62
SVM+N-gram	49.76	49.76	33.06
SVM+LDA	47.28	47.28	33.94
LSTM	50.02	7.69	5.13
BiLSTM	93.68	93.68	93.50
MCDF	97.63	89.74	92.64
TextCNN	94.76	94.76	93.58
ASTNN	95.95	95.21	95.71
TextGCN	89.39	89.05	89.05
PSTDF	99.32	99.32	99.33

4.6. Experiment 5

Our method is not only applicable to Java but also applicable to classification tasks in other languages such as C/C++. Therefore, we designed Experiment 5. This experiment uses an OJ dataset written in C/C++ to test. The samples in this dataset are collected from the online judge system (OJ) published in [15]. The OJ dataset contains a total of 52,000 operational use cases including 104 types of C++ functions. We use this experiment to prove that our method is also applicable to code vulnerability classification scenarios of other programming languages.

Experiment parameters: For PSTDF, we change the parser that parses the source code as AST from javalang to pycparser [34], with the remaining parameters unchanged. For program dependency graph (PDG)-based graph embedding, in this experiment, we also tested the performance of the method based PDG and gated graph neural network (GGNN), this method use Frama-C5 [35] to get the programs' PDGs and then send them to the GGNN for graph embedding. The parameters of the other methods are consistent with Experiment 3.

The experimental results are shown in Table 7.

Table 7. C++ code classification experiment.

Method	Accuracy (%)	Recall (%)	F1 (%)
SVM+TD-IDF	85.45	85.45	85.56
SVM+N-gram	84.66	84.66	84.99
SVM+LDA	0.79	0.008	0.001
TextGCN	79.16	78.31	78.21
PDG+GGNN	79.61	79.61	79.74
TBCNN	94.01	94.01	94.14
ASTNN	97.22	97.29	97.23
PSTDF	97.70	97.73	97.72

Result analysis: It can be seen that PSTDF is also applicable to vulnerability classification scenarios of other programming languages by using the abstract syntax tree parser of the corresponding programming language.

5. Discussion

5.1. Effects of PSTDF

Through the above experiments, we have determined three advantages of PSTDF: (a) It can effectively reduce the loss of semantic structure information when processing abstract syntax trees. (b) It can reduce the negative impact of irrelevant redundant information in AST. (c) In addition, PSTDF also has good generalization ability, it can be applied to vulnerability classification scenarios of other programming languages by changing the corresponding AST parser for different languages.

5.2. Privacy Issues and Limitations

However, our model is temporarily unable to handle the classification of cross-file vulnerabilities such as deserialization vulnerabilities. Another risk is privacy issues: collecting and uploading private data to the centralized server without enough regulation [36]. In practical applications, some organizations or individuals may not want their source code to be uploaded when conducting vulnerability detection, thus creating a risk of privacy disclosure.

6. Conclusions and Future Works

In this paper, we proposed an effective deep forest based on pruned statement tree (PSTDF) to learn the representation of source code. PSTDF performs breadth-first traversal on the AST to obtain a sequence of pruned statement trees, and then encodes each pruned statement tree in sequence to get a sequence of vectors. PSTDF solves the problem of loss of semantic structure information in AST and the problem of how to eliminate irrelevant information in AST to avoid overfitting. We use three datasets, OWASP, SARD, and OJ, to evaluate the performance of PSTDF. The experimental results show that PSTDF is superior to the existing methods, and PSTDF is not only applicable to code vulnerability classification scenarios for Java but also can be applied to code vulnerability classification scenarios for other programming languages by simply modifying the AST parser.

In the future, we will consider federated learning using the distributed normal form cooperative training model to solve privacy issues [37]. Additionally, we will study how to solve the classification problem of cross-file vulnerabilities.

Author Contributions: Methodology, J.D.; software, W.F.; validation, J.D.; formal analysis, J.D.; investigation, W.F.; writing—original draft preparation, W.F.; writing—review and editing, J.D. and L.J.; supervision, L.J.; project administration, L.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (62262034, 62262035).

Data Availability Statement: Publicly available datasets were analyzed in this study. These data and the model source code can be found here: <https://github.com/WeikFu/PSTDF> (accessed on 16 December 2022).

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

Abbreviations

The following abbreviations are used in this manuscript:

J2EE	Java 2 Platform Enterprise Edition
PSTDF	Pruned Statement Tree-based Deep Forest
CVE	Common Vulnerabilities and Exposures
AST	Abstract Syntax Tree
ASTNN	AST-based Neural Network
RNN	Recurrent Neural Networks
BiLSTM	Bi-directional Long Short-Term Memory
LSTM	Long Short-Term Memory
GAN	Generative Adversarial Network
SAE	Sparse Auto Encoder
SMGA	Semantic-complete Graph
GSC	Graph-embedded Semantic Completion
GRU	Gated Recurrent Unit
ST	Statement Tree
PST	Pruned Statement Tree
CNN	Convolutional Neural Network
SVM	Support Vector Machines
MCDF	Malicious Code classification method based on Deep Forest
LDA	Linear Discriminant Analysis
TF-IDF	Term Frequency–Inverse Document Frequency
GCN	Graph Convolutional Networks
OWASP	Open Web Application Security Project
SARD	Software Assurance Reference Dataset

References

1. CVE Details. Available online: <https://www.cvedetails.com/browse-by-date.php> (accessed on 18 December 2022).
2. Younis, A.; Malaiya, Y.; Anderson, C.; Ray, I. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016; pp. 97–104.
3. Anbiya, D.R.; Purwarianti, A.; Asnar, Y. Vulnerability detection in php web application using lexical analysis approach with machine learning. In Proceedings of the 2018 5th International Conference on Data and Software Engineering (ICoDSE), Mataram, Indonesia, 7–8 November 2018; pp. 1–6.
4. Kim, S.; Zhao, J.; Tian, Y.; Chandra, S. Code prediction by feeding trees to transformers. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 150–162.
5. Liang, J.; Wang, M.; Zhou, C.; Wu, Z.; Jiang, Y.; Liu, J.; Liu, Z.; Sun, J. PATA: Fuzzing with Path Aware Taint Analysis. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; IEEE Computer Society: Los Alamitos, CA, USA, 2022; pp. 154–170.
6. Lin, B.; Wang, S.; Wen, M.; Mao, X. Context-aware code change embedding for better patch correctness assessment. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2022**, *31*, 1–29. [[CrossRef](#)]
7. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 783–794.
8. Meng, Y.; Liu, L. A deep learning approach for a source code detection model using self-attention. *Complexity* **2020**, *2020*, 5027198. [[CrossRef](#)]
9. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
10. Liu, G.; Guo, J. Bidirectional LSTM with attention mechanism and convolutional layer for text classification. *Neurocomputing* **2019**, *337*, 325–338. [[CrossRef](#)]
11. Hua, W.; Liu, G. Transformer-based networks over tree structures for code classification. *Appl. Intell.* **2022**, *52*, 8895–8909. [[CrossRef](#)]

12. Xing, Y.; Qian, X.; Guan, Y.; Zhang, S.; Zhao, M.; Lin, W. Cross-project Defect Prediction Method Using Adversarial Learning. *J. Softw.* **2022**, *33*, 2097–2112.
13. Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. Generative adversarial networks. *Commun. ACM* **2020**, *63*, 139–144. [[CrossRef](#)]
14. Bui, D.Q.N.; Yu, Y.; Jiang, L. TreeCaps: Tree-based capsule networks for source code processing. In Proceedings of the 35th AAAI Conference on Artificial Intelligence, Virtual Conference, 2–9 February 2021; pp. 2–9.
15. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
16. Peng, Z.; Yao, Y.; Xiao, B.; Guo, S.; Yang, Y. When urban safety index inference meets location-based data. *IEEE Trans. Mob. Comput.* **2018**, *18*, 2701–2713. [[CrossRef](#)]
17. Kang, J.; Xiong, Z.; Niyato, D.; Zou, Y.; Zhang, Y.; Guizani, M. Reliable federated learning for mobile networks. *IEEE Wirel. Commun.* **2020**, *27*, 72–80. [[CrossRef](#)]
18. Li, W.; Liu, X.; Yuan, Y. SIGMA: Semantic-complete Graph Matching for Domain Adaptive Object Detection. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, New Orleans, LA, USA, 18–24 June 2022; pp. 5291–5300.
19. Li, W.; Liu, X.; Yao, X.; Yuan, Y. SCAN: Cross Domain Object Detection with Semantic Conditioned Adaptation. In Proceedings of the AAAI, Virtual, 22 February–1 March 2022; Volume 6, p. 7.
20. Ye, Z.B.; Yan, B. Survey of Symbolic Execution. *Comput. Sci.* **2018**, *45*, 28–35. (In Chinese with English abstract)
21. Zhou, Z.H.; Feng, J. Deep forest. *Natl. Sci. Rev.* **2019**, *6*, 74–86. [[CrossRef](#)] [[PubMed](#)]
22. Bahdanau, D.; Cho, K.; Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv* **2014**, arXiv:1409.0473.
23. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
24. Barchi, F.; Parisi, E.; Urgese, G.; Ficarra, E.; Acquaviva, A. Exploration of convolutional neural network models for source code classification. *Eng. Appl. Artif. Intell.* **2021**, *97*, 104075. [[CrossRef](#)]
25. Lu, X.; Duan, Z.; Qian, Y.; Zhou, W. Malicious Code Classification Method Based on Deep Forest. *J. Softw.* **2020**, *31*, 1454–1464.
26. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y.; De Vel, O.; Montague, P. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3289–3297. [[CrossRef](#)]
27. Zaremba, W.; Sutskever, I. Learning to execute. *arXiv* **2014**, arXiv:1410.4615.
28. Zhou, P.; Shi, W.; Tian, J.; Qi, Z.; Li, B.; Hao, H.; Xu, B. Attention-based bidirectional long short-term memory networks for relation classification. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), Berlin, Germany, 7–12 August 2016; pp. 207–212.
29. Kim, Y. Convolutional Neural Networks for Sentence Classification. *arXiv* **2014**, arXiv:1408.5882.
30. Yao, L.; Mao, C.; Luo, Y. Graph convolutional networks for text classification. In Proceedings of the AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; Volume 33, pp. 7370–7377.
31. Baygin, M. Classification of text documents based on Naive Bayes using N-Gram features. In Proceedings of the 2018 International Conference on Artificial Intelligence and Data Processing (IDAP), Malatya, Turkey, 28–30 September 2018; pp. 1–5.
32. Wang, W.; Guo, B.; Shen, Y.; Yang, H.; Chen, Y.; Suo, X. Twin labeled LDA: A supervised topic model for document classification. *Appl. Intell.* **2020**, *50*, 4602–4615. [[CrossRef](#)]
33. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. Deep learning similarities from different representations of source code. In Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, 28–29 May 2018; pp. 542–553.
34. Ruberg, P.; Meinberg, E.; Ellervee, P. Software Parser and Analyser for Hardware Performance Estimations. In Proceedings of the 2022 International Conference on Electrical, Computer and Energy Technologies (ICECET), Prague, Czech Republic, 20–22 July 2022; pp. 1–6.
35. Garion, C.; Hattenberger, G.; Pollien, B.; Roux, P.; Thirioux, X. A Gentle Introduction to C Code Verification Using the Frama-C platform. ISAE-SUPAERO; ONERA—The French Aerospace Lab; ENAC, 2022. Available online: <https://hal.science/hal-03625208/> (accessed on 6 January 2023)
36. Feng, J.; Rong, C.; Sun, F.; Guo, D.; Li, Y. PMF: A privacy-preserving human mobility prediction framework via federated learning. *Proc. AcM Interact. Mob. Wearable Ubiquitous Technol.* **2020**, *4*, 1–21. [[CrossRef](#)]
37. Chen, Z.; Yang, C.; Zhu, M.; Peng, Z.; Yuan, Y. Personalized Retrogress-Resilient Federated Learning Toward Imbalanced Medical Data. *IEEE Trans. Med. Imaging* **2022**, *41*, 3663–3674. [[CrossRef](#)] [[PubMed](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.