Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.041949

ARTICLE

Software Vulnerability Mining and Analysis Based on Deep Learning

Shibin Zhao*, Junhu Zhu and Jianshan Peng

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

*Corresponding Author: Shibin Zhao. Email: b1b4b0@163.com

Received: 12 May 2023 Accepted: 11 August 2023 Published: 15 August 2024

ABSTRACT

In recent years, the rapid development of computer software has led to numerous security problems, particularly software vulnerabilities. These flaws can cause significant harm to users' privacy and property. Current security defect detection technology relies on manual or professional reasoning, leading to missed detection and high false detection rates. Artificial intelligence technology has led to the development of neural network models based on machine learning or deep learning to intelligently mine holes, reducing missed alarms and false alarms. So, this project aims to study Java source code defect detection methods for defects like null pointer reference exception, XSS (Transform), and Structured Query Language (SQL) injection. Also, the project uses open-source Javalang to translate the Java source code, conducts a deep search on the AST to obtain the empty syntax feature library, and converts the Java source code into a dependency graph. The feature vector is then used as the learning target for the neural network. Four types of Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM), Bi-directional Long Short-Term Memory (BiLSTM), and Attention Mechanism + Bidirectional LSTM, are used to investigate various code defects, including blank pointer reference exception, XSS, and SQL injection defects. Experimental results show that the attention mechanism in two-dimensional BLSTM is the most effective for object recognition, verifying the correctness of the method.

KEYWORDS

Vulnerability mining; software security; deep learning; static analysis

1 Introduction

In today's information age, the emergence of computers relieves people's pressure, especially in work and life. The emergence of computers has had a significant impact on both personal and professional aspects of life. It has alleviated pressures by enhancing productivity, enabling efficient communication, providing access to vast information resources, automating tasks, facilitating remote work, promoting connectivity, enabling convenience in daily tasks, and offering entertainment and leisure options. It not only promotes the development of human technology but also has a huge impact on the world. Computer technology is always being updated, and the rapid development of its software technology has led to an endless stream of related security vulnerabilities. The rapid development of computer software technology has contributed to the emergence of new security vulnerabilities due to increased complexity, evolving attack vectors, faster development cycles, inadequate security practices, and inadequate testing and validation processes. Since 2017, the number of recorded vulnerabilities has shown a jumping increase [1], which has attracted the attention of relevant technical personnel, and further illustrates the development of technology for exploiting vulnerabilities in software has reached a critical moment. As shown in Fig. 1.

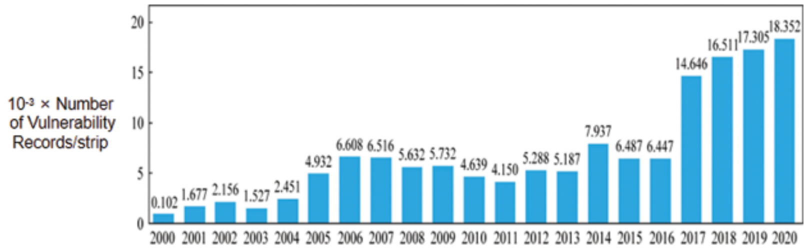


Figure 1: The number of vulnerability records disclosed by the US National Vulnerability Database (NVD) over the years

Due to various reasons, such as the limitation of the development language, the programming ability and experience of the software developer, and even small negligence of the software developer, loopholes may occur. Skilled and experienced developers are less likely to introduce vulnerabilities due to their knowledge of secure coding practices, while less experienced developers may be more prone to making coding mistakes that lead to vulnerabilities. Therefore, software vulnerabilities cannot be avoided. Once these vulnerabilities are discovered and exploited by attackers, they may cause irreparable losses. For example, in October 2016, attackers used the IoT botnet to carry out a distributed denial of service (Distributed denial of service attack, DDoS) attack on the US Internet domain name resolution service provider Dyn, resulting in the failure of a large number of website domain names to be correctly resolved and the network paralyzed [2]. In August 2020, hackers discovered a smart contract vulnerability on the DeFi platform cryptocurrency decentralized exchange (Opyn) and launched an attack, causing a loss of approximately US$370,000 [3]. Therefore, actively digging out the loopholes in the software and repairing them in time can effectively prevent the harm caused by the loopholes.

At present, software engineering is developing rapidly, and a large number of software and tools have been developed to meet the various needs of users. On the one hand, security personnel have limited energy and cannot take into account all new products; on the other hand, this software is large in scale, complex in calling relationships, and have high code coverage. rate of return, and the cost is high. Vulnerability mining through manual analysis or vulnerability rules extracted by experts is no longer suitable for today's software development. Following the wave of the artificial intelligence era, many security researchers apply deep learning to the field of software security vulnerability mining, conduct in-depth analysis and extraction of vulnerability characteristics automatically on massive samples, dig out the vulnerabilities that are most likely to be attacked and repair them in time, from Cut off the possibility of vulnerability exploitation at the source.

The software vulnerability mining model based on deep learning mainly studies and improves the data representation technology for extracting vulnerability feature information, and retains the features related to vulnerability information to the greatest extent to improve the accuracy of vulnerability mining. The key steps in the data collection stage of software vulnerability mining based on deep learning are Identifying data sources, define vulnerability criteria, Collect the dataset, Label the data, Pre-process the data, Split the dataset, Ensuring data quality, Augment the dataset (optional), Store and organize the data. Currently, vulnerability characterization technologies are mainly divided into sequence-based characterization technology, syntax tree-based characterization technology, graph-based characterization technology, text-based characterization technology, and hybrid characterization technology. Fig. 2 shows the vulnerabilities of these five characterization methods in the past 10 years. Mining the proportion of [4].

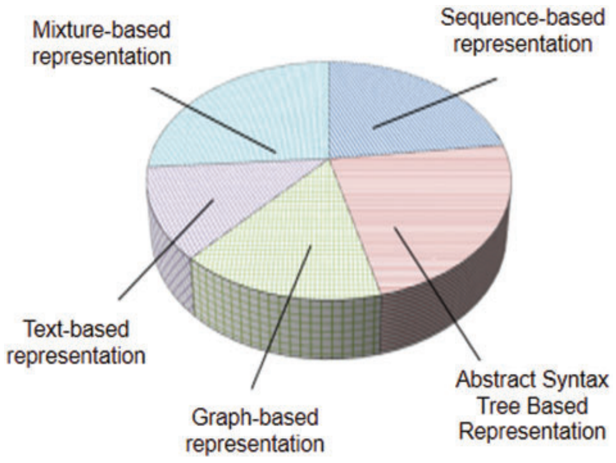


Figure 2: Proportion of research on software vulnerability mining based on different code representations

The sequence-based characterization method is to perform lexical analysis on the program source code or binary file to obtain program running information, such as source code identifiers, keywords, function names and other important information, and convert the source code into a Token sequence flow. Literature applied deep learning to sequence information extraction for the first time [5]. According to the relationship between data flow or program flow and code semantic relationship, a heuristic-based code gadget statement set was constructed, and samples were mapped to feature vectors and input into a bidirectional long-term short-term memory network (BLSTM), experiments show that VulDeePecker can achieve a lower false positive rate than other vulnerability detection systems while freeing human experts from the tedious work of manually defining features. However, it is only for C/C++ source code and there are too few use case vulnerabilities, so the false positive rate is high. Secondly, VulDeePecker is a binary classification system, which can only detect whether there are vulnerabilities in the source code but cannot detect the type of vulnerability. Literature uses control dependencies and local feature information based on literature to propose code attention for the detection of vulnerability types and builds a multi-classification vulnerability detection system uVulDeePecker [6]. Experiments show that code attention has a good detection effect in a small sample space, and the ability to detect multi-category vulnerabilities can be improved by using control dependencies. However, it can only detect vulnerabilities related to API calls, and cannot locate the specific location of the vulnerabilities.

The representation method based on abstract syntax is to use ASTExtractor, Javalang, ANTLR and other open-source tools to convert the source code file into an abstract syntax tree (abstract syntax tree, AST), and obtain the hierarchy by traversing and extracting the node information of the syntax tree Characterized information. AST provides a natural program representation at the function level and preserves more information about the source code compared to control flow graphs (CFGs), which usually do not contain variable declarations [7–9]. Software vulnerabilities are usually reflected in the grammatical structure of the source code. Literature proposed a data processing method based on an abstract syntax tree to extract all grammatical features and reduce data redundancy [10]. In addition, the packet filling method is applied to the bidirectional gated recurrent unit (BGRU) network to train variable-length data without truncation and padding. Experiments show that the model can predict vulnerabilities with high accuracy and a low false positive rate, but if the vulnerability involves multiple functions or multiple files, it is difficult for this framework to extract features from it. Literature combines the convolutional neural network (CNN) with the long short-term memory (LSTM) network to extract local features and global features and uses the attention mechanism to determine the weight of each element in the feature space. Furthermore, the method rewrites the source code and converts it into vectors without the guidance of specific domain knowledge. Experiments on the buffer errors dataset (CWE119) and resource management errors dataset (CWE-399) show that the method achieves remarkable detection performance [11]. This research [12] addresses the challenges faced by distributed teams in global software development by predicting risks related to time, cost, and resources. Neural network approaches, including Bayesian Regularization, were used to assess the impact of these factors, with Bayesian Regularization demonstrating strong performance based on the MSE criterion.

The graph-based representation method is to convert the source code into a data dependency graph (data dependency graph, DDG), control dependency graph (Control Dependence Graph, CDG), program dependency graph (Program Dependence Graph, PDG), etc. The relationship between element variables and function variables contains more grammatical and semantic information. Then, using graph embedding technology and deep neural network for learning and training, good detection results can be obtained. Graph embedding technology has achieved good detection results by efficiently representing graph structures in low-dimensional vectors, enabling effective analysis and detection of patterns, anomalies, or vulnerabilities in complex graph-based data. Literature uses the graph representation method to automatically extract the program dependency graph from the source code, combines the GNN model to extrapolate the vulnerability, and then generates a recommendation list through the cosine similarity, that is, the security audit system that may have the same vulnerability. The graph representation method used to extract the Program Dependence Graph (PDG) involves creating a directed graph where nodes represent program statements or expressions, and edges represent the dependencies between them. Control dependencies represent the order of execution, and data dependencies capture the flow of data between statements. The accuracy rate can reach 99.2% on the Juliet dataset [13]. Literature deduplicated the repeatedly compiled feature vectors of the source code, combined with the system dependency graph to generate the minimum intermediate representation, and then converted the intermediate code into a vector that can retain structural and semantic information by expanding the corpus, and finally input it into CNN to obtain A high-level signature representation of a vulnerability. However, it can only be used to detect static vulnerabilities, and cannot detect vulnerabilities in compiled software [14].

The text-based representation method uses word frequency statistics, word2vec and other methods to map the feature words extracted from the text information of the source code, assembly instructions, and code after lexical analysis, and then inputs them into the neural network model for training. The literature considers that vulnerability codes can be obtained by submitting vulnerability repairs, and proposes an automatic vulnerability submission identification tool based on Hierarchical Attention Network (HAN) to expand the existing vulnerability data set [15]. By submitting vulnerability repairs, researchers can quickly locate Vulnerable code. However, the construction of the dataset is time-consuming and labour-intensive [16]. Literature uses text mining and machine learning to extract knowledge from open-source codes to classify and structure the source codes. Studying JAVA files in open-source projects reveals that useful patterns can be extracted from the source code, and the proposed technique is applied to SQL injection detection [17]. Reference developed a code generator s-bAbI capable of generating any number of code samples of controlled complexity. It contains syntactically valid C programs with non-trivial control flow, and safe and unsafe buffer writes marked at the line-of-code level. This code is simple compared to real code, but the performance of the static analyzer is not as expected. Literature regards the given binary code as a sequence of machine instructions, and then uses the variational autoencoder (VAE) theory to develop a maximum divergence sequential autoencoder (MDSAE), which can calculate the representation of the binary code form to maximize the separation of vulnerable and non-vulnerable binaries for vulnerability detection, while still retaining the critical information inherent in the original binary [18].

Hybrid-based characterization technology combines at least two feature representation methods among the four characterization technologies of sequence, syntax tree, graph, and text to deeply mine the feature information of vulnerabilities. Features related to vulnerability information that improve the accuracy of vulnerability mining include code patterns, contextual information, severity assessment, vulnerable functions/APIs analysis, dependencies analysis, historical data, vulnerability type classification, metadata (CVE), patch notes analysis, and exploitability information. Compared with single-representation technology, the hybrid-based vulnerability mining system can synthesize grammatical and semantic information, identifier keywords, architecture, program attributes, program operation information, etc., and obtain feature vectors with a strong correlation with vulnerabilities, which is helpful to Improve detection performance. The literature proposes a general-purpose graph neural network model based on Devign to perform graph-level classification by learning rich code semantic representations [19]. It includes a new Conv module that can efficiently extract useful features from learned rich node representations for graph-level classification. The literature proposes an automatic detection method for software function-level source code vulnerabilities based on graph representation learning. First, the code is converted into a Simplified Code Property Graph (SCPG), which can preserve the syntax and semantic information of the source code while maintaining Small enough computing space. Then use graph neural network and multi-layer perceptron to learn graph representation and automatically extract features, saving the workload of feature engineering. Experimental results show that the method can not only effectively mine the loopholes, but also take into account the mining efficiency [20]. This paper [21] conducts a comprehensive review of recent studies focusing on deep learning for vulnerability detection. It identifies four significant advancements that have revolutionized the field and brought new perspectives. These key breakthroughs introduce innovative ideas and approaches, injecting new life into the domain. The survey summarizes the progress and trends in vulnerability detection, emphasizing the importance of further research contributions in this rapidly expanding area. Researchers have used machine learning and deep learning to automatically detect vulnerabilities to mitigate their negative effects on information security, the economy, social stability, and national security. They propose CNN-LSTM, a composite neural network with long short-term memory. The results show a 95% increase in F1-score, precision, recall, and accuracy. CNN-LSTM outperforms other deep learning models in false-positive, miss, and accuracy rates [22].

The problem statement and the main contribution of the paper are discussed as follows: Software vulnerabilities have increased due to computer software's rapid development. These vulnerabilities endanger user privacy and property. Security defect detection methods that rely on manual analysis or expert reasoning miss detections and have high false positive rates. To address this issue, Machine learning and deep learning have been used to intelligently identify vulnerabilities, reducing missed alarms and false alarms. This project examines Java source code defects like null pointer reference exceptions, XSS (Transform), and SQL injections. The project uses the open-source Javalang tool to translate Java source code and performs an in-depth search on the Abstract Syntax Tree (AST) to extract an empty syntax feature library. The neural network learns from the dependency graph of the Java source code. CNN, LSTM, BiLSTM, and LSTM neural networks investigate null pointer reference exceptions, XSS, and SQL injections.

The overall structure of the paper is discussed as follows: This paper comprises five sections. The first section covers the introduction of this research study. Related work to research has been explained in Section 2; the research methodology of the proposed work is discussed in Section 3 and its sub-sections; Section 4 represents the experimental verification; finally, the conclusion of the proposed work is discussed in Section 5.

2 Related Works

This article analyzes the characteristics and causes of software vulnerabilities and then introduces in detail the open-source tools that convert source code into AST syntax trees and program dependency graphs.

2.1 Software Vulnerability

Software security breaches are one of the root causes of network intrusions. In information security, a software vulnerability is a weakness or defect in a system that makes the system sensitive to a specific threat attack or dangerous event or has the possibility of an attack threat [23].

Software vulnerabilities are mainly caused by the lack of programming ability and security awareness of software developers. Although developers can be trained to improve their programming ability and security awareness, the occurrence of vulnerabilities is still unavoidable [24]. Researchers focus on improving software reliability by developing fewer complex applications. They propose a novel method that combines the analytic hierarchy method (AHP), hesitant fuzzy sets, and the TOPSIS technique for reliability prediction. This approach applies to various failure datasets and software applications, enabling estimation of reliability based on software type. However, further research is necessary to identify the most optimal model for accurate reliability prediction [25]. According to the causes of software vulnerabilities, software vulnerabilities can be divided into memory corruption, logic errors, input validation, design errors, and configuration errors. Once these vulnerabilities are exploited by attackers, they may damage the confidentiality, integrity and availability of information systems.

Software vulnerabilities usually have the following characteristics:

(1) Persistence and timeliness. Once a vulnerability is created, it will not disappear automatically until it is discovered and fixed. As time goes by, new vulnerabilities are constantly being created, so the problem of software vulnerabilities is always there.

(2) broad and specific. Vulnerabilities exist widely in various software and are inevitable, and different types of software, different versions of hardware, and different system configurations may cause different vulnerability problems [26].

(3) Availability and concealment. Security holes generally do not affect the normal use of software and are difficult to be discovered by security personnel. But once it is discovered and exploited by the attacker, it will cause huge economic losses [27–29].

2.2 *Abstract Syntax Tree Analysis*

Abstract Syntax Tree (abstract syntax tree, hereinafter referred to as AST) is one of the most common methods to represent source code in the form of tree structure. Abstract syntax tree analysis is based on lexical analysis and syntax analysis to extract the architecture of the program. The nodes near the root node mainly contain relatively rough information, such as functions, classes, etc.; the information contained in the nodes becomes fine-grained as the depth of the node deepens, and the last leaf node may correspond to a certain variable or constant.

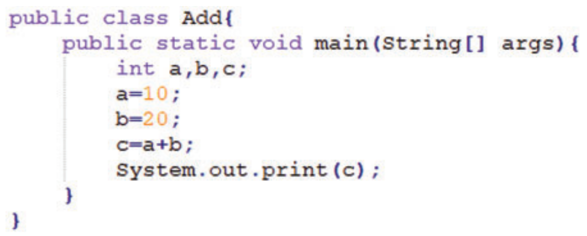
The open-source tool used in this paper to convert the source code into an abstract syntax tree is Kang [30]. It is a library in Python for processing Java source code, which can map every grammatical structure to a class object. The tool first uses a lexical analyzer to perform lexical analysis on the source code, generates a token sequence flow and checks whether the token sequence flow conforms to the Java language specification, and then inputs it into the syntax analyzer to generate an abstract syntax tree.

2.3 *Program Dependency Graph Analysis*

A program dependency graph is an intermediate program representation that contains the data dependencies and control dependencies of each operation in the program. Data dependencies are used only to represent the relative data flow relationships of a program. Control dependencies are introduced to represent only the basic control flow relationships of programs in a similar fashion. Control dependencies are usually inferred from control flow graphs [31].

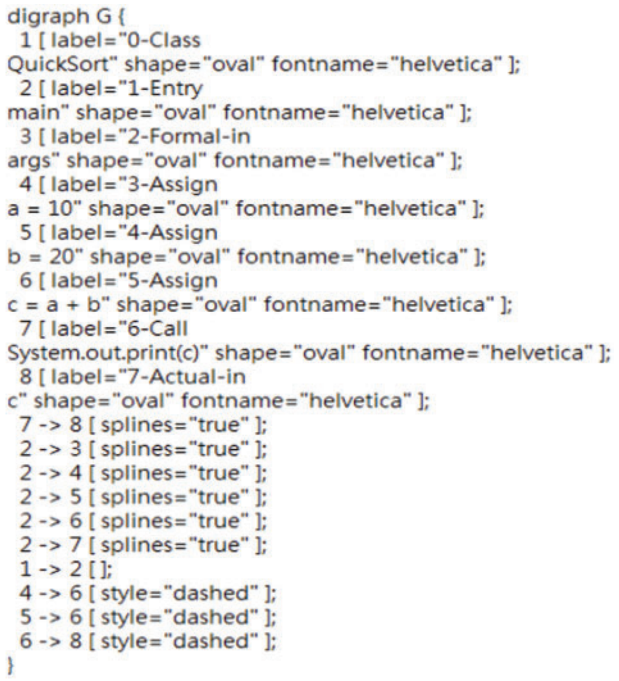
The open-source tool used in this paper to convert the source code into a program dependency graph is sourcedg [32]. It is a framework for generating program dependency graphs from source code based on transformation rules. It first takes Java source code as input and outputs a parsed program dependency graph. Then define the formal semantics of the graph generated based on the conversion rules, construct the control dependency graph and control flow graph of the program according to these rules, and finally analyze the data flow to obtain the data dependency. Figs. 3 and 4 show a simple Java source code fragment, Fig. 3 shows a text document converted using sourcedg, and Fig. 5 shows the corresponding program dependency graph. In Fig. 5, the nodes represent statement information, and the edges represent dependency information. The solid lines represent control dependencies, and the dotted lines represent data dependencies [33].

Fig. 3 shows a simple Java source code snippet.



```
public class Add{
    public static void main(String[] args){
        int a,b,c;
        a=10;
        b=20;
        c=a+b;
        System.out.print(c);
    }
}
```

Figure 3: Java source code snippet



```
digraph G {
 1 [ label="0-Class
QuickSort" shape="oval" fontname="helvetica" ];
 2 [ label="1-Entry
main" shape="oval" fontname="helvetica" ];
 3 [ label="2-Formal-in
args" shape="oval" fontname="helvetica" ];
 4 [ label="3-Assign
a = 10" shape="oval" fontname="helvetica" ];
 5 [ label="4-Assign
b = 20" shape="oval" fontname="helvetica" ];
 6 [ label="5-Assign
c = a + b" shape="oval" fontname="helvetica" ];
 7 [ label="6-Call
System.out.print(c)" shape="oval" fontname="helvetica" ];
 8 [ label="7-Actual-in
c" shape="oval" fontname="helvetica" ];
 7 -> 8 [ splines="true" ];
 2 -> 3 [ splines="true" ];
 2 -> 4 [ splines="true" ];
 2 -> 5 [ splines="true" ];
 2 -> 6 [ splines="true" ];
 2 -> 7 [ splines="true" ];
 1 -> 2 [ ];
 4 -> 6 [ style="dashed" ];
 5 -> 6 [ style="dashed" ];
 6 -> 8 [ style="dashed" ];
}
```

Figure 4: Sourcedg converted text document

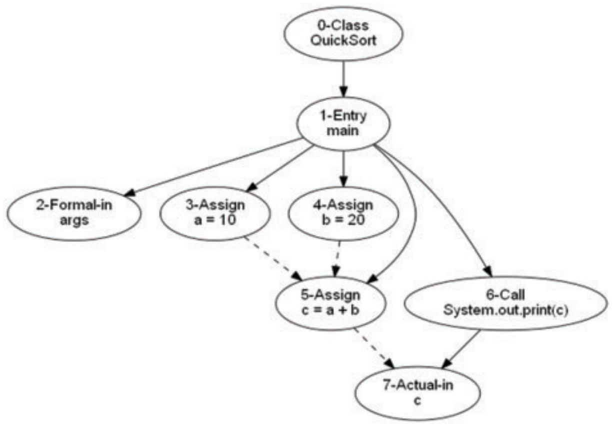


Figure 5: Program dependency graph for code segments

2.4 *BiLSTM Structure*

LSTMs can predict a word by examining the relevance of the nearest information elements. However, for preceding words such as, they can only capture the i-th word's dependencies. For many NLP tasks, examining the preceding information is often not sufficient to perform accurate predictions, the following words, such as pairs of words, may also be useful. To obtain the dependence of the words around the word, the forward LSTM and the backward LSTM are combined to form a BiLSTM [34]. The bidirectional network can capture the forward information and the reverse information at the same time, making the use of text information more comprehensive and the effect better. The BiLSTM neural network model processes the sequence in both forward and backward directions, capturing context information effectively. The BiLSTM structure is shown in Fig. 6.

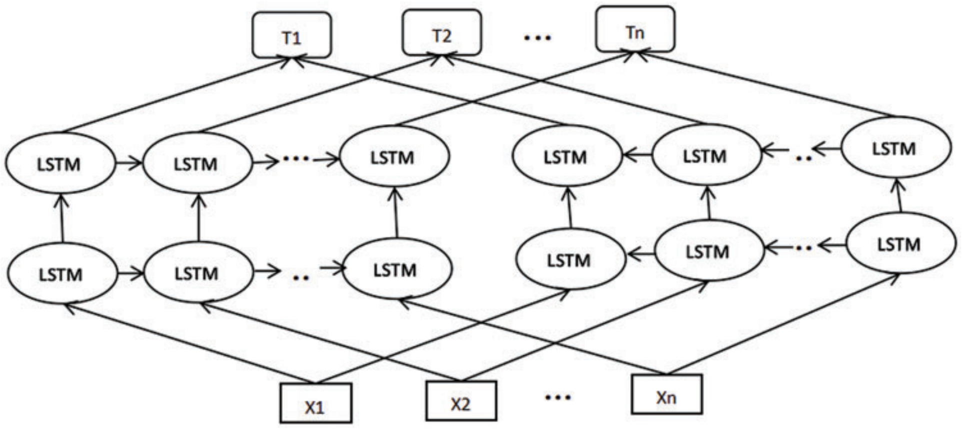


Figure 6: BiLSTM model structure

2.5 Attention Mechanism

Attention can be said to have become one of the most important concepts in the field of deep learning. It was inspired by human biological systems, which tend to specialize in unique parts when processing large amounts of information. With the development of deep neural networks, attention mechanisms have been widely used in different application domains [35]. Deep neural networks and attention mechanisms are widely used in various application domains, including computer vision for tasks like image recognition and object detection, natural language processing (NLP) for tasks such as language translation and sentiment analysis, speech recognition for voice-based interfaces, and recommendation systems for personalized recommendations in e-commerce and content platforms. The attention mechanism enables the model to focus on relevant parts of the input sequence by assigning importance weights. The main reason for introducing the attention mechanism in the model of this paper is that the context vector can be calculated according to the attention distribution, and the grammatical semantics and structural information can be further preserved. The attention mechanism structure is shown in Fig. 7.

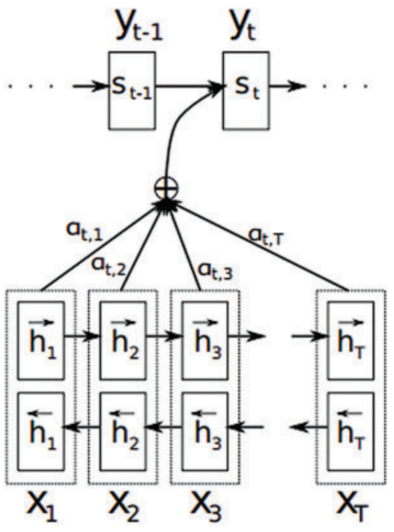

y_{t-1}
y_t
s_{t-1}
s_t
$a_{t,1}$
$a_{t,2}$
$a_{t,3}$
$a_{t,T}$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_3$ $\overrightarrow{h}_T$
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_3$ $\overleftarrow{h}_T$
X_1 X_2 X_3 X_T

Figure 7: Attention mechanism model structure

3 Research Methodology

The software vulnerability mining method based on deep learning constructed in this paper mainly includes three stages: data collection, learning and detection. The data collection stage mainly collects a large amount of labelled vulnerability sample data for deep learning; the learning stage first cleans and integrates the collected data, then extracts features by combining syntax trees and graphs, maps them to vector space, and finally inputs them into attention The mechanism and BiLSTM are trained; the detection stage is to use the trained neural network model to predict the target code program. This chapter will focus on the feature extraction method and the construction of the neural network model.

3.1 Vulnerability Mining Work Framework

The vulnerability mining work framework proposed in this paper mainly includes three stages of data collection, learning and detection.

Data collection stage: mainly to collect Java source codes containing vulnerabilities, and at the same time, there must be clear label information for each Java source code, that is, not only need to indicate whether there are vulnerabilities but also indicate the type of vulnerabilities for files with vulnerabilities. The types of vulnerabilities that need to be described for Java source code files with vulnerabilities may include SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Command Injection, Remote Code Execution (RCE), Denial of Service (DoS), and Unvalidated Input. After searching a large number of data sets, the benchmark and Juliet data sets are currently eligible, so these two data sets are fused as the data set of this project.

Learning phase: First, clean up the collected benchmark and Juliet datasets, merge Java source codes belonging to the same type of vulnerabilities, and then remove duplicate and redundant data. Second, balance the number of vulnerability samples and positive and negative vulnerability sample data to avoid deviations. Then, feature extraction is performed on the data set, and after normalization, word2vec vectorization is used to obtain feature vectors, which are finally input into the attention mechanism and BiLSTM neural network model for training.

Detection stage: After the model is generated, another part of the data is needed to test the accuracy of the model. The unknown software system can be tested only after the test is qualified. The process is consistent with the above stage. After the vectorized representation, the data is passed the models produced during the characterization phase are tested to see their final predictive accuracy. It involves assessing the model's performance on unseen data to determine its ability to make accurate predictions and generalize well to new instances.

3.2 Vulnerability Signature Matching

First, use Javalang to convert the Java source code containing null pointer reference exception, XSS (cross-site scripting) and SQL injection vulnerabilities into an abstract syntax tree, and then judge whether the node elements contain pointer information or grammatical features of sensitive characters to form vulnerability features candidate set. Next, we will show how to extract candidate sets of vulnerability features from the abstract syntax trees of the three types of vulnerabilities: null pointer reference exception, XSS (cross-site scripting) and SQL injection. The flow chart of generating the vulnerability feature candidate set is shown in Fig. 8.

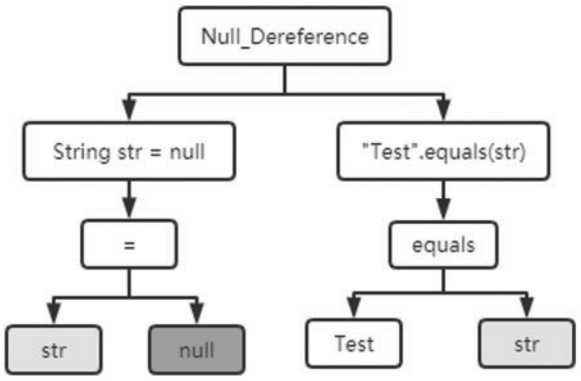


Figure 8: Pointer exception syntax features

As shown in Fig. 8, this is a Java code fragment that uses a null pointer to perform illegal operations. It is not difficult to see that there is a null pointer exception in this code, but no error will be reported during program compilation and operation. A null pointer exception can occur in the code without reporting any errors during compilation if the null reference is not properly checked or handled in the code. Scenarios that can lead to this situation include uninitialized variables, improper null value checks, and passing null references to methods that do not handle them correctly. And what this article gives is just a very simple example, the null pointer exception, in reality, is much more complicated and hidden. What is shown is the corresponding pointer exception syntax feature. As an example, if $E_{i,j,z}$ the identifier for the declaration, is initialized to null and there is no subsequent assignment (that is, str), put it into the vulnerability feature candidate set of null pointer exceptions.

As shown in Fig. 9, this is a code fragment with SQL injection vulnerability. The statement is an important interface for Java database operations. The Statement interface in Java facilitates the execution of SQL statements on a connected database by providing methods to create and execute SQL queries and updates. It acts as a channel for sending SQL commands to the database and retrieving the results. It is used to send SQL statements to be executed to the database based on established database connections. The name is a parameter obtained from the front end. Some recommended approaches for mitigating SQL injection vulnerabilities in code handling SQL statements embedded in user input include Parameterized queries/prepared statements, Input validation/sanitization, Principle of least privilege, Use of stored procedures/Object Relational Mapping (ORM) frameworks and Regular database system updates.

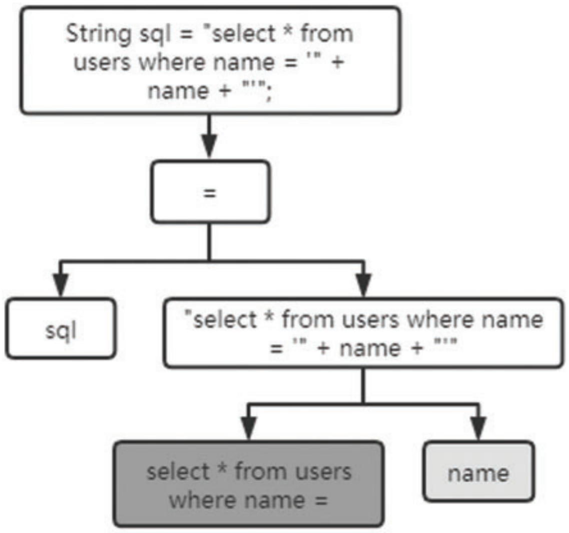

String sql = "select * from users where name = '" + name + "'";
=
sql
"select * from users where name = '" + name + "'"
select * from users where name =
name

Figure 9: Syntactic features of SQL injection vulnerabilities

It is not difficult to see that the name input by the user is not filtered here. If the attacker enters maliciously, it is "admin'or 1 = 1", You can bypass the identity authentication system to log in and mine sensitive information in the database. Fig. 10 shows the corresponding syntax features of SQL injection vulnerabilities. As an example, it is a sensitive point containing SQL statements, put into the candidate set of SQL injection vulnerability characteristics.

This code fragment returns the user request and displays it on the front-end page; Fig. 11 is the back-end code fragment for data processing, and the content is the data transmitted from the front-end and model. Add attribute is the data that responds to the front end. The potential security vulnerability described in the code fragments is Cross-Site Scripting (XSS). In addition, the potential consequences of a successful XSS attack include unauthorized access to sensitive user information, session hijacking, spreading of malicious code or links, defacement of web pages, and potential manipulation or theft of data. It is not difficult to see that the backend directly returns the requested data to the front without any filtering, and there is a reflective cross-site scripting attack. Content Security Policy (CSP) is a security mechanism that helps prevent cross-site scripting (XSS) attacks by specifying and enforcing restrictions on the resources that a web page can load. It defines a policy that restricts the types of content that can be executed or displayed on a web page, reducing the risk of executing malicious scripts injected by attackers. If accessing content = 2, the page will display 2, thus realizing the attack. Fig. 11 shows the corresponding XSS vulnerability syntax features. As an example, if $E_{i,j,z}$ It is a sensitive point that contains XSS, put it into the XSS vulnerability feature candidate set.

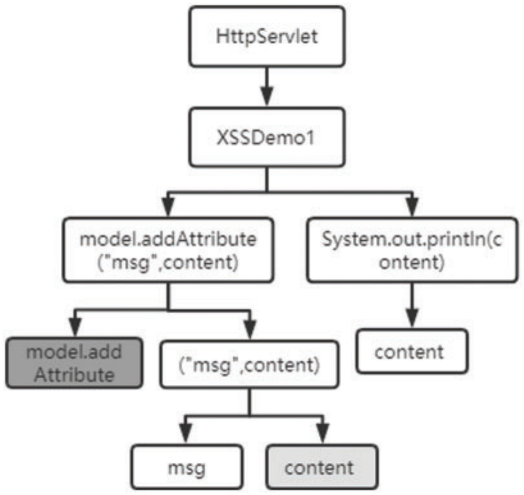


Figure 10: XSS vulnerability grammatical features

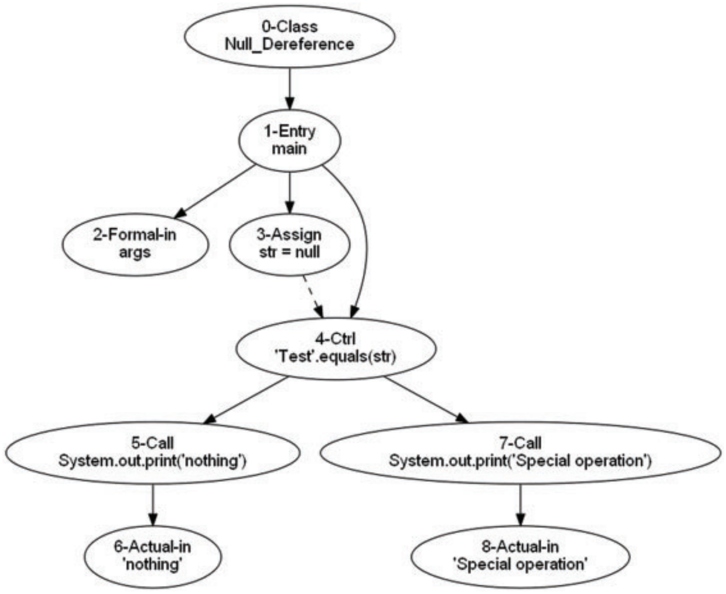

0-Class Null_Dereference
1-Entry main
2-Formal-in args
3-Assign str = null
4-Ctrl 'Test'.equals(str)
5-Call System.out.print('nothing')
7-Call System.out.print('Special operation')
6-Actual-in 'nothing'
8-Actual-in 'Special operation'

Figure 11: Program dependency graph of null pointer reference exception

3.3 Generate Vulnerability Code Slices

It mainly introduces the key technology and main steps of slicing according to the vulnerability feature candidate set. First, use the open-source tool sourcedg to convert the Java source code into a program flowchart, and then generate forward slices and backward slices according to data dependencies, control dependencies, and vulnerability feature candidate sets [33]. Then, the forward slice and the backward slice are fused, and the repeated part is removed to generate a program slice, which constitutes a vulnerability feature set.

Due to the complexity of SQL injection vulnerabilities and XSS cross-site scripting vulnerabilities, the generated program dependency graph has many nodes and edges, and the control dependencies and data dependencies are complicated. SQL Injection includes Manipulating SQL queries to inject malicious code, bypass input validation, and gain unauthorized database access. Then, XSS (Cross-Site Scripting) involve exploiting improper input sanitization to inject and execute malicious scripts in the victim's browser, potentially leading to data theft or unauthorized actions. Due to limited space, this article takes null pointer reference exception as an example to illustrate the detailed process of slice generation vulnerability signature code. Slice generation in vulnerability signature code involves extracting relevant code fragments or slices that are associated with a particular vulnerability. These code slices capture the specific patterns, logic, or functions that are vulnerable to attacks. Those challenges in vulnerability signature code generation include code variability, false positives/negatives, emerging vulnerabilities, and context/environment dependencies. They are addressed through analyzing patterns, testing, collaboration, and regular updates to adapt to new vulnerabilities and their variations. The program dependency graph of the null pointer reference exception of the source code fragment is shown in Fig. 11.

To put it simply, forward slicing is the collection of all node statements that can be reached by functional dependencies and data dependencies starting from the elements of the vulnerability feature candidate set on the program dependency graph; backward slicing is any node statement that can be reached on the program dependency graph. These dependencies provide insights into the relationships and interactions between different parts of the code, allowing for the identification of potential vulnerabilities and their associated features. Forward slicing starts from a given program point and identify all the statements that may be affected by its values or dependencies. On the other hand, backward slicing starts from a given program point and identifies all the statements that may have influenced its values or dependencies. A collection of node statements that rely on functional dependencies and data dependencies to reach the elements of the vulnerability feature candidate set and use this as the endpoint. In the null pointer exception part, put str into the vulnerability feature candidate set, taking Fig. 10 as an example, there is no forward slice, backward slices are: 'Test'.equals (str), System.out.print ("nothing"), nothing, System. out.print ("Special operation"), Special operation. Then the forward and backward slices are fused to generate program slices and put into the vulnerability feature set. The fusion of forward slice and backward slice combines the results of both slicing techniques to identify and remove repeated code segments, generating a concise program slice. This fusion process considers both forward and backward dependencies, eliminating redundancy and focusing on relevant code portions for analysis or debugging purposes.

3.4 Deep Neural Network Model Learning

The input of the two-way long-short-term memory network based on the attention mechanism cannot directly be the vulnerability feature set, so each sample in the vulnerability feature set needs to be tokenized, and then word2vec is used to vectorize the token flow. The attention mechanism is a component of neural networks that helps focus on specific parts of the input during processing, but it does not directly capture the features or characteristics of vulnerabilities. Tokenization is necessary for the input of a two-way Long Short-Term Memory (LSTM) network based on the attention mechanism because it breaks down the text into smaller units (tokens) to represent the input sequence. This enables the model to process and understand the sequential information in the text effectively. Since user-defined variables and function names are determined according to personal programming habits and are complex and changeable, each sample needs to be standardized before tokenizing the vulnerability feature set. Standardization before tokenization in the vulnerability feature set may include removing comments, normalizing variable names, handling preprocessor directives, resolving library-specific functions, and addressing language-specific syntax.

In other words, the variables in the sample are uniformly represented by "value1", "value2", etc., and the function names are uniformly represented by "function1", "function2", etc., "function2" and so on. By doing so, the structure and contextual information can be preserved to a large extent. Then, the statements of each sample in the vulnerability feature set are tokenized by lexical analysis, i.e., "str=null" is converted into a token containing 3 tokens ("str", "=", "null") into a sequence containing 3 tokens ("str", "=", "null"). Finally, word2vec is used for word embedding, where each "word" of the input sequence is represented as a dense vector of fixed dimension. The continuous Bag-of-Words (CBOW) model of word2vec is applied to convert the vector of each element into a 100-word embedding dimension. The Continuous Bag-of-Words (CBOW) model of word2vec is a neural network-based approach for generating word embeddings. It predicts a target word based on its context words within a given window. The CBOW model aims to learn a continuous representation of words by maximizing the probability of predicting the target word given its context words. With word2Vec, the elements of the vector can be represented by semantically meaningful vector representations, so that elements sharing similar contexts in the code can be located close together in the vector space. The final one-dimensional numerical sequence of each sample is transformed into a two-dimensional matrix feature input to the neural network for training. Converting a one-dimensional sequence into a two-dimensional matrix helps capture the contextual relationships between elements in the sequence, enabling the model to consider the order and dependencies of the data points more effectively.

In this paper, a bidirectional long and short-term memory network based on an attention mechanism is used to train the feature vectors, and its network model structure consists of an input layer, a bidirectional LSTM layer, an attention layer, a softmax layer and an output layer. The overall structure is shown in Fig. 12.

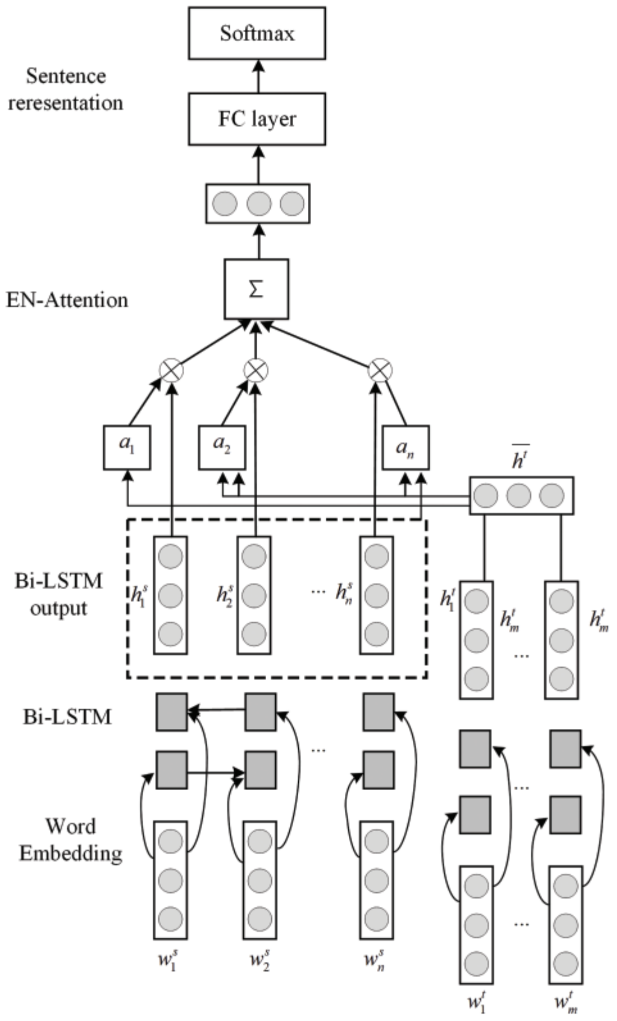


Figure 12: Model structure

4 Experimental Verifications

This paper describes the experimental results. To find the best deep learning model for the framework proposed in this paper, four different neural network models (CNN, LSTM, BiLSTM, attention mechanism + bidirectional LSTM) are used to detect different types of code vulnerabilities (Null pointer reference exceptions, XSS (cross-site scripting) and SQL injection vulnerabilities) for learning, and compare the accuracy of the detection results of each model and other indicator. The accuracy of detection results can be compared using indicators like precision, recall, F1-score, or ROC curve to evaluate different aspects of the model's performance such as precision/recall balance, overall accuracy, and discriminative power.

4.1 Experimental Data Set Selection

For the selection of test data sets, this article needs to collect Java source codes that contain null pointer reference exceptions, XSS (cross-site scripting) and SQL injection vulnerabilities. At the same time, there must be clear label information for each Java source code, that is, not only Indicate whether there are loopholes or not, and the files with loopholes shall also indicate the type of loopholes. After searching a large number of data sets, the benchmark and Juliet data sets are currently eligible, so these two data sets are fused as the data set of this project. Among them, CWE476 (null pointer exception vulnerability) has 693 samples, CWE83 (cross-site scripting vulnerability) has 1620 samples, and CWE89 (SQL injection vulnerability) has 1480 samples. CWE476 (Null Pointer Dereference) refers to instances where a program attempts to use a null pointer, leading to runtime errors and crashes. CWE79 (Cross-Site Scripting) represents vulnerabilities that allow attackers to inject and execute malicious scripts in web pages, leading to unauthorized actions, data theft, and website compromise. CWE89 (SQL Injection) denotes vulnerabilities that enable attackers to manipulate SQL queries through unvalidated or unsanitized user input, potentially leading to unauthorized access, data breaches, or database compromise. The proportion of each vulnerability is shown in Table 1.

Table 1: The proportion of each vulnerability in the data set

Vulnerability number	CWE476		CWE83		CWE89	
Vulnerability name	Null pointer exception vulnerability		Cross-site scripting vulnerability		SQL injection vulnerability	
Number of vulnerabilities	693		1620		1480	
	There are loopholes	No loopholes	There are loopholes	No loopholes	There are loopholes	No loopholes
	325	368	800	820	752	728

4.2 Experiment Settings

4.2.1 Neural Network Parameter Settings

There are a total of 693 null pointer exception vulnerabilities, of which 368 have no vulnerabilities and 325 have vulnerabilities. The specific parameters of its neural network model are shown in Table 2.

Table 2: Null pointer exception related vulnerability experimental parameters

Parameter name	Parameter value
Optimizer adam	Lr = 0.002
Lossless function	categorical_crossentropy
Metrics	Accuracy
Batch size	10
Epochs	10

There are 1480 XSS vulnerabilities, of which there are 752 without vulnerabilities and 728 with vulnerabilities. The recommended approaches for identifying XSS vulnerabilities during security testing and code reviews include thorough input validation, output encoding, context validation, using security scanning tools, performing manual code reviews, conducting penetration testing, and following secure coding guidelines. The specific parameters of its neural network model are shown in Table 3:

Table 3: XSS related vulnerability experimental parameters

Parameter name	Parameter value
Optimizer adam	Lr = 0.002
Loss function	categorical_crossentropy
Metrics	Accuracy
Batch size	50
Epochs	10

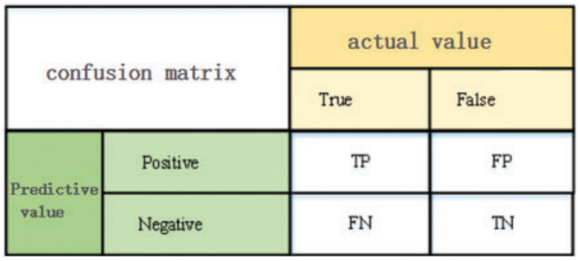
There are a total of 1620 SQL injection vulnerabilities, of which 820 have no vulnerabilities and 800 have vulnerabilities. The specific parameters of its neural network model are shown in Table 4:

Table 4: SQL injection-related vulnerability experimental parameters

Parameter name	Parameter value
Optimizer adam	Lr = 0.002
Loss function	categorical_crossentropy
Metrics	Accuracy
Batch size	65
Epochs	10

4.2.2 Evaluation Index Introduction

For the evaluation of model quality, this paper uses precision rate, recall rate, F1 value and ROC curve as the evaluation indicators of the Java source code vulnerability detection framework based on syntax trees and graphs. The four widely used metrics for evaluating classification models are accuracy, precision, recall and F1-score. These four metrics are widely used in the evaluation of classification models. First, the number of positive and negative classes between the predicted value and the real value is listed as a confusion matrix as shown in Fig. 13.



confusion matrix		actual value	
		True	False
Predictive value	Positive	TP	FP
	Negative	FN	TN

Figure 13: Confusion matrix

Accuracy (*Precision*) is defined as the Eq. (1):

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

Recall rate (*Recall*) is defined as the Eq. (2):

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

The precision rate is the ratio of the model correctly classified as a positive class to all samples classified as a positive class by the model, and the recall rate is defined as the ratio of the correct classification as a positive class to all actual positive classes, and the F1 value is as in Eq. (3). It is a compromise between the precision rate and the recall rate, which can more objectively evaluate the pros and cons of a model, and prevent abnormal valuations caused by excessive deviations in the number of positive and negative samples.

$$F1 = 2 \cdot \frac{Precison \cdot Recall}{Precision + Recall} \tag{3}$$

receiver operating characteristic curve (Receiver Operating Characteristic, ROC).

Different from the above three indicators, ROC does not care about specific scores but pays more attention to the scores between positive and negative samples. ROC curve analysis focuses on the relationship between true positive rate and false positive rate, rather than specific scores because it measures the classifier's ability to distinguish between positive and negative samples regardless of the specific threshold used for classification. It is often used when there is a large difference in the number of positive and negative samples. Its meaning is to randomly select a target member and non-target members, the probability that the target members get higher scores under this identification model than the selected non-target members.

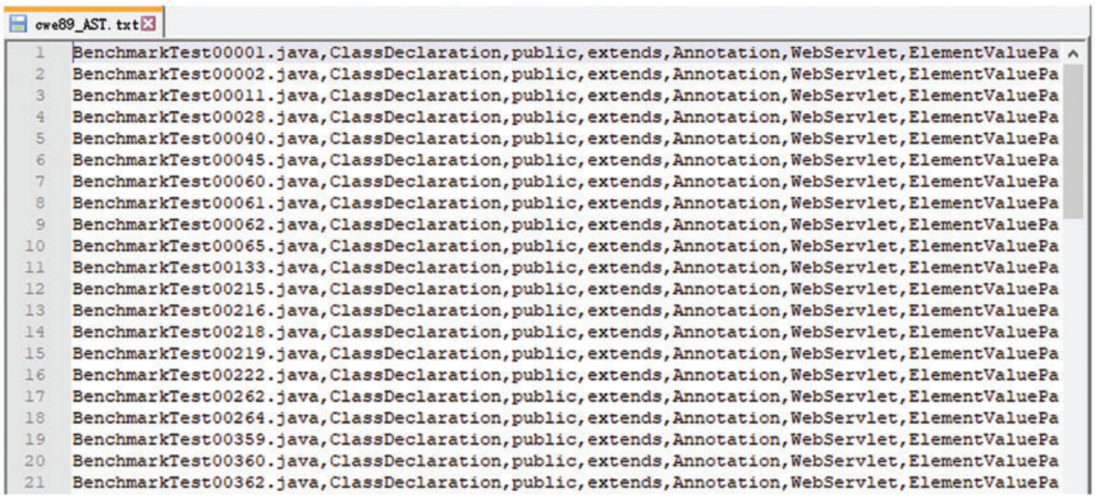
The area under the ROC curve (Area under ROC curve, AUC) is the numerical representation of this indicator. The ROC curve is generally located above the straight-line y = x, so the range of the AUC value is generally between 0.5 and 1. The effect of an algorithm model is positively correlated with its corresponding AUC value. The closer the AUC value is to 1, the It shows that the effect of the classification model is better.

4.3 Test Process

After cleaning the data set of null pointer reference exception, XSS (cross-site scripting) and SQL injection vulnerability, use Java lang to convert it into an abstract syntax tree (to see the structure of the abstract syntax tree more intuitively, where the nodes of the syntax tree The information is extracted, as shown in Fig. 14). The functions of the three vulnerabilities are Null Pointer Reference Exception which occurs when a program tries to use a null object reference, resulting in runtime errors and program crashes, XSS (Cross-Site Scripting) where attackers inject and execute malicious scripts in web pages, enabling unauthorized actions, data theft, and website defacement. Then, SQL Injection where attackers manipulate SQL queries through user input, leading to unauthorized access, data manipulation, and potential database compromise.

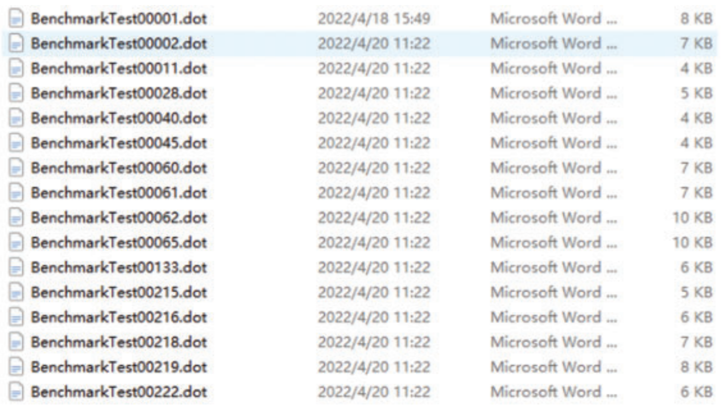
Use sourcedg to convert the program dependency graph (as shown in Fig. 15). Perform program slicing to obtain a set of vulnerability features, then standardize each sample in the set and convert it into a token stream (as shown in Fig. 16), and use word2vec to obtain feature vectors (as shown in

Fig. 17). Finally, it is input into the bidirectional LSTM neural network model based on the attention mechanism for training (as shown in Figs. 18–20).



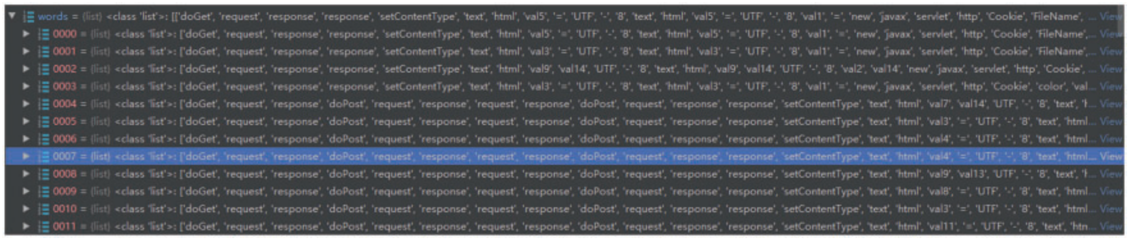
```
cwe89_AST.txt
BenchmarkTest00001.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00002.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00011.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00028.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00040.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00045.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00060.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00061.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00062.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00065.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00133.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00215.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00216.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00218.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00219.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00222.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00262.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00264.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00359.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00360.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
BenchmarkTest00362.java,ClassDeclaration,public,extends,Annotation,WebServlet,ElementValuePa
```

Figure 14: CWE476_AST node information



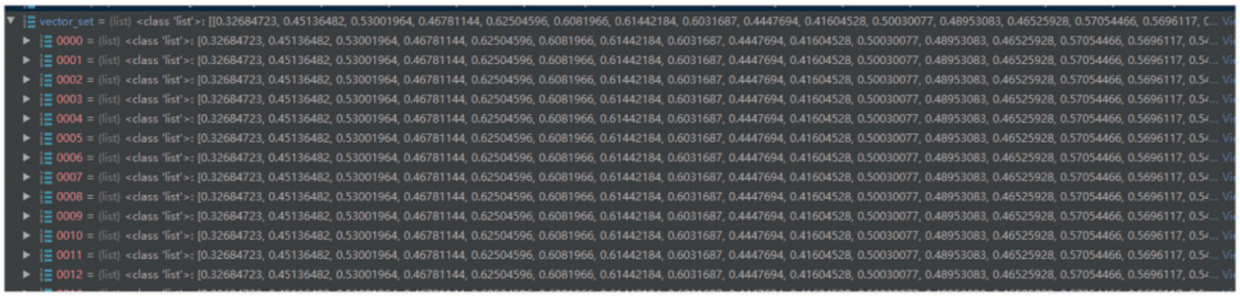
```
BenchmarkTest00001.dot   2022/4/18 15:49   Microsoft Word ...    8 KB
BenchmarkTest00002.dot   2022/4/20 11:22   Microsoft Word ...    7 KB
BenchmarkTest00011.dot   2022/4/20 11:22   Microsoft Word ...    4 KB
BenchmarkTest00028.dot   2022/4/20 11:22   Microsoft Word ...    5 KB
BenchmarkTest00040.dot   2022/4/20 11:22   Microsoft Word ...    4 KB
BenchmarkTest00045.dot   2022/4/20 11:22   Microsoft Word ...    4 KB
BenchmarkTest00060.dot   2022/4/20 11:22   Microsoft Word ...    7 KB
BenchmarkTest00061.dot   2022/4/20 11:22   Microsoft Word ...    7 KB
BenchmarkTest00062.dot   2022/4/20 11:22   Microsoft Word ...   10 KB
BenchmarkTest00065.dot   2022/4/20 11:22   Microsoft Word ...   10 KB
BenchmarkTest00133.dot   2022/4/20 11:22   Microsoft Word ...    6 KB
BenchmarkTest00215.dot   2022/4/20 11:22   Microsoft Word ...    5 KB
BenchmarkTest00216.dot   2022/4/20 11:22   Microsoft Word ...    6 KB
BenchmarkTest00218.dot   2022/4/20 11:22   Microsoft Word ...    7 KB
BenchmarkTest00219.dot   2022/4/20 11:22   Microsoft Word ...    8 KB
BenchmarkTest00222.dot   2022/4/20 11:22   Microsoft Word ...    6 KB
```

Figure 15: CWE476 program dependency graph



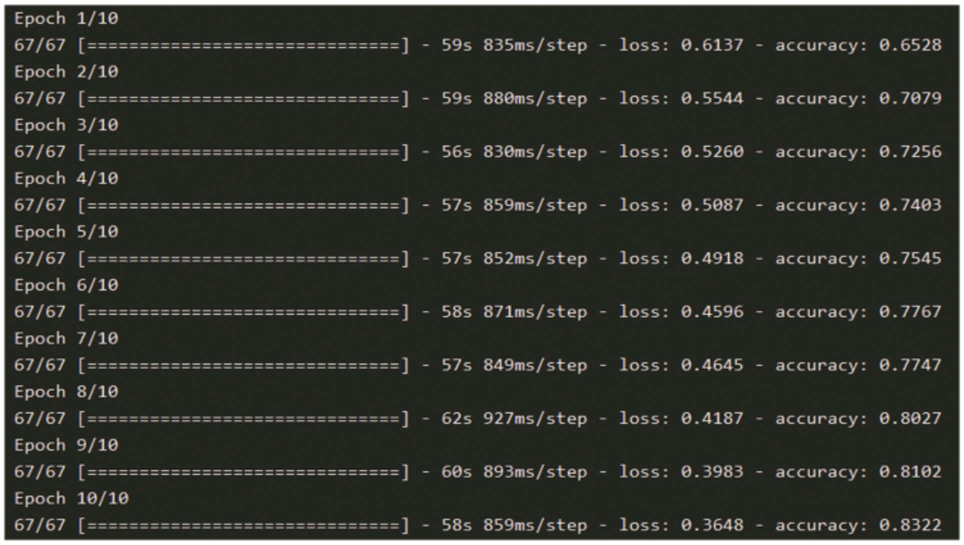
```
words = (list) <class 'list'>: [['doGet', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val5', '=', 'UTF', '-', '8', 'text', 'html', 'val5', '=', 'UTF', '-', '8', 'val1', '=', 'new', 'javax', 'servlet', 'http', 'Cookie', 'FileName', ... View
  0000 = (list) <class 'list'>: ['doGet', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val5', '=', 'UTF', '-', '8', 'text', 'html', 'val5', '=', 'UTF', '-', '8', 'val1', '=', 'new', 'javax', 'servlet', 'http', 'Cookie', 'FileName',... View
  0001 = (list) <class 'list'>: ['doGet', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'val1', '=', 'new', 'javax', 'servlet', 'http', 'Cookie', 'FileName',... View
  0002 = (list) <class 'list'>: ['doGet', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val9', 'val14', 'UTF', '-', '8', 'text', 'html', 'val9', 'val14', 'UTF', '-', '8', 'val2', 'val14', 'new', 'javax', 'servlet', 'http', 'Cookie', ... View
  0003 = (list) <class 'list'>: ['doGet', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'val1', '=', 'new', 'javax', 'servlet', 'http', 'Cookie', 'color', 'val... View
  0004 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val7', 'val14', 'UTF', '-', '8', 'text', 'I... View
  0005 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'text', 'html... View
  0006 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val4', '=', 'UTF', '-', '8', 'text', 'html... View
  0007 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val4', '=', 'UTF', '-', '8', 'text', 'html... View
  0008 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val9', 'val13', 'UTF', '-', '8', 'text', 'I... View
  0009 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val8', '=', 'UTF', '-', '8', 'text', 'html... View
  0010 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val3', '=', 'UTF', '-', '8', 'text', 'html... View
  0011 = (list) <class 'list'>: ['doGet', 'request', 'response', 'doPost', 'request', 'response', 'request', 'response', 'doPost', 'request', 'response', 'response', 'setContentType', 'text', 'html', 'val11', '=', 'UTF', '-', '8', 'text', 'htn... View
```

Figure 16: CWE476 token flow



```
▼ vector_set = (list) <class 'list'>: [[0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, C... Vi
  ▶ 0000 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0001 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0002 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0003 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0004 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0005 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0006 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0007 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0008 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0009 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0010 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0011 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
  ▶ 0012 = (list) <class 'list'>: [0.32684723, 0.45136482, 0.53001964, 0.46781144, 0.62504596, 0.6081966, 0.61442184, 0.6031687, 0.4447694, 0.41604528, 0.50030077, 0.48953083, 0.46525928, 0.57054466, 0.5696117, 0.54... Vi
```

Figure 17: Word2vec after the eigenvector



```
Epoch 1/10
67/67 [==============================] - 59s 835ms/step - loss: 0.6137 - accuracy: 0.6528
Epoch 2/10
67/67 [==============================] - 59s 880ms/step - loss: 0.5544 - accuracy: 0.7079
Epoch 3/10
67/67 [==============================] - 56s 830ms/step - loss: 0.5260 - accuracy: 0.7256
Epoch 4/10
67/67 [==============================] - 57s 859ms/step - loss: 0.5087 - accuracy: 0.7403
Epoch 5/10
67/67 [==============================] - 57s 852ms/step - loss: 0.4918 - accuracy: 0.7545
Epoch 6/10
67/67 [==============================] - 58s 871ms/step - loss: 0.4596 - accuracy: 0.7767
Epoch 7/10
67/67 [==============================] - 57s 849ms/step - loss: 0.4645 - accuracy: 0.7747
Epoch 8/10
67/67 [==============================] - 62s 927ms/step - loss: 0.4187 - accuracy: 0.8027
Epoch 9/10
67/67 [==============================] - 60s 893ms/step - loss: 0.3983 - accuracy: 0.8102
Epoch 10/10
67/67 [==============================] - 58s 859ms/step - loss: 0.3648 - accuracy: 0.8322
```

Figure 18: CWE476 neural network model training

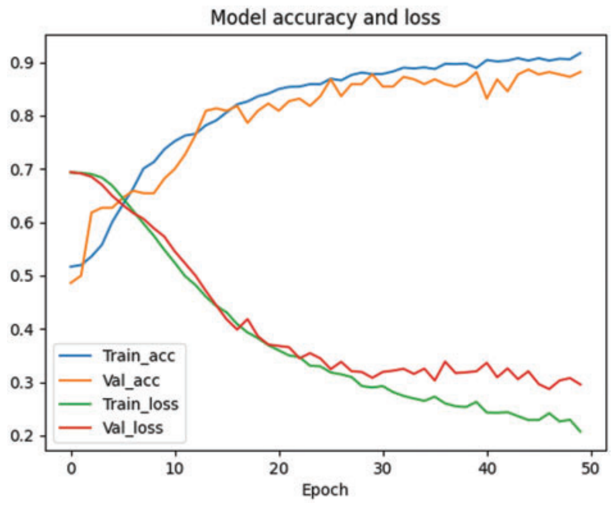
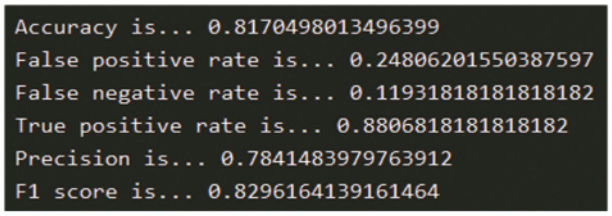


Figure 19: CWE476 neural network model training loss and ACC curve



```
Accuracy is... 0.8170498013496399
False positive rate is... 0.24806201550387597
False negative rate is... 0.11931818181818182
True positive rate is... 0.8806818181818182
Precision is... 0.7841483979763912
F1 score is... 0.8296164139161464
```

Figure 20: CWE476 neural network model (BiLSTM) test results

4.4 Comparison of Model Results

To find the best deep learning model for the framework proposed in this paper, four different neural network models (CNN, LSTM, BiLSTM, attention mechanism + bidirectional LSTM) were used to detect different types of code vulnerabilities (null pointer reference exception, XSS (Cross-site scripting) and SQL injection vulnerability) for learning, the accuracy rate (Accuracy), false positive rate (FPR), FNR, TPR, precision rate (Precision), and F1-score of the training results are shown in Tables 5 and 6.

Table 5: Null pointer exception-related vulnerability experiment results

Evaluation index	CNN	LSTM	BiLSTM	Attention mechanism + Two-way LSTM
Accuracy	0.7577	0.7863	0.8170	0.8388
FPR	0.4375	0.3406	0.2481	0.2645
FNR	0.0516	0.0894	0.1193	0.060
TPR	0.9484	0.9105	0.8807	0.9398
Precision	0.6893	0.7323	0.7841	0.7842
F1-score	0.7983	0.8117	0.8296	0.8551

Table 6: Experimental results of SQL injection-related vulnerabilities

Evaluation index	CNN	LSTM	BiLSTM	Attention mechanism + Two-way LSTM
Accuracy	0.8531	0.8484	0.8560	0.8850
FPR	0.1591	0.2441	0.1909	0.0597
FNR	0.1346	0.0610	0.0980	0.1703
TPR	0.8653	0.9389	0.9019	0.8297
Precision	0.8445	0.7973	0.8286	0.9328
F1-score	0.8548	0.8623	0.8637	0.8782

As can be seen from the above Tables 5–7, the attention mechanism + bidirectional LSTM has the best detection effect for the three types of vulnerabilities: null pointer reference exception, XSS (cross-site scripting) and SQL injection vulnerability. The efficient use of the attention mechanism and bidirectional LSTM in achieving the best detection effect involves leveraging the attention mechanism to assign weights to different parts of the input sequence, emphasizing relevant information. Among them, the vulnerability detection effect of null pointer exception is the worst, which may be due to the small sample size, which cannot effectively learn the vulnerability characteristics.

Table 7: XSS experimental results of cross-site scripting-related vulnerabilities

Evaluation index	CNN	LSTM	BiLSTM	Attention mechanism + Two-way LSTM
Accuracy	0.8469	0.8690	0.9090	0.9203
FPR	0.0877	0.1753	0.0884	0.1076
FNR	0.2184	0.0876	0.0934	0.0515
TPR	0.7816	0.9124	0.9065	0.9484
Precision	0.8988	0.8418	0.9109	0.8979
F1-score	0.8361	0.8757	0.9087	0.9225

5 Conclusion

This paper presents a Java source code vulnerability detection framework using syntax trees and graphs to detect null pointer reference exceptions, XSS (cross-site scripting), and SQL injection vulnerabilities. The framework converts Java source code into an AST syntax tree, which is then processed to obtain semantic information. The program dependency graph is then processed, with the set of program slices normalized and vectorized to preserve structure and context information. The feature vector is then input into a neural network model for training. Four different neural network models (CNN, LSTM, BiLSTM, attention mechanism + bidirectional LSTM) were used to detect different types of code vulnerabilities, with bidirectional BLSTM based on attention mechanism being the best. However, the framework has some shortcomings, such as its limited detection of null pointer reference exceptions, XSS (cross-site scripting), and SQL injection vulnerabilities. The set of sensitive points of statements of different vulnerability types directly affects the detection accuracy, and the static detection method has defects. To address these issues, the next step is to extend the detection method to other vulnerabilities and increase the number of types it can detect. Additionally, enriching the sensitive point collection of each vulnerability type can improve the accuracy of vulnerability detection. Finally, a combination of dynamic detection and static detection modules is designed to reduce false negative and false positive rates of vulnerability mining.

Acknowledgement: None.

Funding Statement: This work is supported by the Provincial Key Science and Technology Special Project of Henan (No. 221100240100).

Author Contributions: All authors contributed to the design and methodology of this study, the assessment of the outcomes and the writing of the manuscript. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: No datasets were generated or analyzed during the current study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] P. Qian, *et al.*, “A survey of research on smart contract security vulnerability detection technology,” *J. Softw.*, vol. 33, no. 8, pp. 3059–3085, 2022.

[2] Y. Chen, Q. Dai, and H. Wu, “Mirai botnet malware analysis and monitoring data research,” *J. Net. Inform. Security*, vol. 3, no. 8, pp. 39–47, 2017.

[3] Y. Wang, “Analysis of hacking incidents in the field of digital currency and encryption,” *Comput. Netw.*, vol. 2020, no. 17, pp. 47, 2020.

[4] M. Gu *et al.*, “Software security vulnerability mining based on deep learning,” *J. Compt. Res. Develop.*, vol. 58, no. 10, pp. 2140–2162, 2021. doi: 10.7544/issn1000-1239.2021.20210620.

[5] Z. Li *et al.,* “VulDeePecker: A deep learning-based system for vulnerability detection,” arXiv preprint arXiv:1801.01681, 2018.

[6] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “μVulDeePecker: A deep learning-based system for multiclass vulnerability detection,” *IEEE Trans. Dependable Secure Comput.*, vol. 3, no. 99, pp. 1, 2019. doi: 10.1109/TDSC.2019.2942930.

[7] T. Liu, “Software vulnerability mining techniques based on data fusion and reverse engineering,” *Wirel. Commun. Mob. Comput.*, vol. 2022, no. 4, pp. 1–6, 2022. doi: 10.1155/2022/4329034.

[8] Y. Cheng, B. Cui, C. Chen, T. Baker, and T. Qi, “Static vulnerability mining of IoT devices based on control flow graph construction and graph embedding network,” *Comput. Commun.*, vol. 197, pp. 267–275, 2023. doi: 10.1016/j.comcom.2022.10.021.

[9] J. Bovet and T. Parr, “ANTLRWorks: An ANTLR grammar development environment,” *Softw. Pract. Exper.*, vol. 38, no. 12, pp. 1305–1332, 2008. doi: 10.1002/spe.872.

[10] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, “Efficient vulnerability detection based on abstract syntax tree and Deep Learning,” in *IEEE INFOCOM, 2020–IEEE Conf. Comput. Commun. Works. (INFOCOM WKSHPS)*, Toronto, ON, Canada, 2020, pp. 722–727.

[11] D. Cao, J. Huang, X. Zhang, and X. Liu, “FTCLNet: Convolutional LSTM with fourier transform for vulnerability detection,” in *2020 IEEE 19th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Guangzhou, China, 2020, pp. 539–546.

[12] A. Iftikhar, M. Alam, R. Ahmed, S. Musa, and M. M. Su’ud, “Risk prediction by using artificial neural network in global software development,” *Comput. Intell. Neurosci.*, vol. 2021, no. 2, pp. 1–25, 2021. doi: 10.1155/2021/2922728.

[13] J. Zeng, X. Nie, L. Chen, J. Li, G. Du and G. Shi, “An efficient vulnerability extrapolation using similarity of graph kernel of PDGs,” in *2020 IEEE 19th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Guangzhou, China, 2020, pp. 1664–1671.

[14] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, “Automated vulnerability detection in source code using minimum intermediate representation learning,” *Appl. Sci.*, vol. 10, no. 5, pp. 1692, 2020. doi: 10.3390/app10051692.

[15] M. Sun, W. Wang, H. Feng, H. Sun, and Y. Zhang, “Identity vulnerability fix commits automatically using hierarchical attention network,” *Secur. Safe.*, vol. 7, no. 23, pp. 17–31, 2018. doi: 10.4108/eai.13-7-2018.164552.

[16] F. Nembhard, M. Carvalho, and T. Eskridge, “Extracting knowledge from open-source projects to improve program security,” in *SoutheastCon 2018*, St. Petersburg, FL, USA, 2018, pp. 1–7.

[17] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, “Towards security defect prediction with AI,” arXiv preprint arXiv:1808.09897, 2018.

[18] T. Le *et al.*, “Maximal divergence sequential autoencoder for binary software vulnerability detection,” in *Int. Conf. on Learning Representations*, 2019.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Adv. Neural Inf. Process. Syst.*, vol. 2019, no. 915, pp. 10197–10207, 2019.

[20] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in C/C++ source code with graph representation learning," in *2021 IEEE 11th Annu. Comput. Commun. Workshop Conf. (CCWC)*, NV, USA, 2021, pp. 1519–1524.

[21] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020. doi: 10.1109/ACCESS.2020.3034766.

[22] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," *IET Software*, vol. 16, no. 5, pp. 516–526, 2022. doi: 10.1049/sfw2.12066.

[23] G. Li, "Source code vulnerability mining method based on graph neural network," *Int. J. Front. Eng. Technol.*, vol. 4, no. 4, pp. 21–32, 2022. doi: 10.25236/IJFET.2022.040404.

[24] G. Koolin, *Research on Vulnerability Detection Method Based on BiLSTM Model*. China: Nanjing University of Aeronautics and Astronautics, 2020.

[25] K. Sahu, F. A. Alzahrani, R. K. Srivastava, and R. Kumar, "Evaluating the impact of prediction techniques: Software reliability perspective," *Comput., Mater. Contin.*, vol. 67, no. 2, pp. 1471–1488, 2021. doi: 10.32604/cmc.2021.014868.

[26] C. Song, J. Huang, and D. Wang, "A survey of computer security vulnerabilities detection technology," *Inf. Net. Security*, vol. 2012, no. 1, pp. 3, 2012.

[27] Z. Li, D. Zou, Z. Wang, and H. Jin, "A survey of source code-oriented static detection of software vulnerabilities," *J. Net. Inf. Security*, vol. 5, no. 1, pp. 1–14, 2019. doi: 10.11959/j.issn.2096-109x.2019001.

[28] J. Liu *et al.*, "A survey of software and network security research," *J. Softw.*, vol. 29, no. 1, pp. 27, 2018.

[29] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Third Int. Conf. on Software Testing, Verification and Validation*, Paris, France, 2010, pp. 421–428.

[30] Z. Kang, "A review on javascript engine vulnerability mining," *J. Phys.: Conf. Ser.*, vol. 1744, no. 4, pp. 042197, 2021. doi: 10.1088/1742-6596/1744/4/042197.

[31] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987. doi: 10.1145/24039.24041.

[32] J. Marin Victor and C. R. Rivero, "Towards a framework for generating program dependence graphs from source code," in *Proc. 4th ACM SIGSOFT Int. Workshop on Software Analytics*, 2018, pp. 30–36.

[33] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, 2021. doi: 10.1109/TDSC.2021.3051525.

[34] Y. Goldberg and O. Levy, "word2vec Explained: Deriving Mikolov et al.'s negative-sampling word-embedding method," arXiv preprint arXiv:1402.3722, 2014.

[35] Z. Niu, G. Zhong, and H. Yu, "A review on the attention mechanism of deep learning," *Neurocomputing*, vol. 452, pp. 48–62, 2021. doi: 10.1016/j.neucom.2021.03.091.