High-Confidence Computing 3 (2023) 100148

Contents lists available at ScienceDirect

High-Confidence Computing

journal homepage: www.sciencedirect.com/journal/high-confidence-computing

Research article

JFinder: A novel architecture for java vulnerability identification based quad self-attention and pre-training mechanism

Jin Wang[1], Zishan Huang[1], Hui Xiao, Yinhao Xiao*

School of Information Science, Guangdong University of Finance and Economics, Guangzhou 510320, China

ARTICLE INFO

Article history:
Received 14 May 2023
Revised 29 May 2023
Accepted 6 July 2023

Keywords:
Software vulnerabilities
Security risks
Vulnerability identification methods
Pre-training
Structural information
Self-attention

ABSTRACT

Software vulnerabilities pose significant risks to computer systems, impacting our daily lives, productivity, and even our health. Identifying and addressing security vulnerabilities in a timely manner is crucial to prevent hacking and data breaches. Unfortunately, current vulnerability identification methods, including classical and deep learning-based approaches, exhibit critical drawbacks that prevent them from meeting the demands of the contemporary software industry. To tackle these issues, we present JFinder, a novel architecture for Java vulnerability identification that leverages quad self-attention and pre-training mechanisms to combine structural information and semantic representations. Experimental results demonstrate that JFinder outperforms all baseline methods, achieving an accuracy of 0.97 on the CWE dataset and an F1 score of 0.84 on the PROMISE dataset. Furthermore, a case study reveals that JFinder can accurately identify four cases of vulnerabilities after patching.

1. Introduction

As modern software continues to increase in functionality, the likelihood of vulnerabilities also grows. These vulnerabilities pose significant risks to cybersecurity, with implications for individuals, the healthcare industry, and industrial production. For individuals, such vulnerabilities can result in the leak of sensitive information, leading to identity theft and fraud. In the healthcare sector, cybersecurity breaches may entail the theft of health information, ransomware attacks on hospitals, and even attacks on implanted medical devices. In industrial production, software vulnerabilities can introduce corresponding vulnerabilities in products reliant on that software, as exemplified by the Log4j2 vulnerability. Apache Log4j2 is susceptible to remote code execution (RCE) attacks,[2] wherein an attacker with the ability to modify logging configuration files can create malicious configurations using the JDBC Appender and a data source referencing a JNDI URI, enabling remote code execution. The Log4j2 vulnerability impacted tens of thousands of products but was swiftly addressed, preventing a catastrophic cyber event.

The identification of vulnerabilities is crucial for ensuring system security and timely remediation of security flaws, thus protecting against hacking and data breaches. However, vulnerability detection can be a laborious and challenging process. Researchers have been continuously exploring methods to automate this task. Initially, researchers manually identified features and employed machine learning to ascertain the existence of vulnerabilities, but this approach proved time-consuming. Subsequently, deep learning techniques were utilized to automatically detect vulnerability features and classify them. Many of these methods involve the use of graph neural networks to identify vulnerability patterns, leading to the development of new graph neural network types (e.g., graph convolution networks, graph attention networks, and graph autoencoders). Some approaches rely on structural information (e.g., abstract syntax tree, control flow graph, and data flow graph) to construct graphs, while others solely employ semantic information of codes for embedding. However, these methods have not demonstrated satisfactory performance on real software vulnerability datasets and remain unsuitable for industrial application.

To address these issues, we propose JFinder, a novel architecture for Java vulnerability identification leveraging structural information with MetaPaths, a quad self-attention layer, and a pre-trained programming language model. We utilize a third-party library to obtain the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) of source code as structural information. We derive the Code Snippet Sequence (CSS) using a pre-trained model, UniXcoder [1], a transformer-based language model designed for natural language processing tasks in the software development domain, trained on an extensive corpus of source code and natural language text related to

* Corresponding author.
E-mail address: 20191081@gdufe.edu.cn (Y. Xiao).

[1] These authors contributed equally to this work.

[2] https://logging.apache.org/log4j/2.x/security.html

https://doi.org/10.1016/j.hcc.2023.100148

software development. UniXcoder enables the conversion of program language into a feature matrix, providing accurate semantic representations of code snippets as it is trained on program language. JFinder then feeds semantic and structural information into convolutional networks and multilayer perceptrons to predict vulnerability presence. Overall, JFinder uniquely combines semantic and structural information to comprehensively analyze the execution process from multiple perspectives. We have implemented JFinder as an open-source project on GitHub.[3] We evaluated JFinder on CWE and PROMISE datasets, comprising a total of 20,402 code snippets, of which 7,355 were vulnerable. Experimental results indicate that JFinder achieved outstanding performance on the CWE dataset, with an accuracy rate of 97%. On the PROMISE dataset, JFinder attained an industrially viable level with F1 scores reaching 0.83. We also conducted case studies with four cases; after vulnerabilities were addressed, JFinder no longer identified these cases as vulnerable. The case study results demonstrated the capacity of JFinder to uncover robust vulnerability patterns and provide insights. Our contributions are threefold:

- We propose a novel architecture for java vulnerability identification, JFinder, which combines structural information and semantic information from a code snippet. We open the source of JFinder in Github.
- We have conducted a large number of experiments and compared them with recent excellent methods. The results show that JFinder outperforms all baseline methods and the results are satisfactory.
- We conduct case studies to explore the robustness and intelligence level of JFinder. Experience results show that JFinder understands the meaning of the vulnerabilities in depth.

The rest of the paper is organized as follows: Section 2 presents the recently advanced background knowledge of our approach. Section 3 details the design and technical components of the JFinder framework. Section 4 demonstrates our implementation of JFinder. Section 5 reports our evaluation results and case studies on JFinder. Section 6 outlines the most related work. Section 7 concludes the paper with a future research discussion.

2. Background

In this section, we present the background knowledge of some recently advanced technologies utilized by JFinder.

2.1. Pre-trained models

Pre-trained models are machine learning models that have already undergone training for certain tasks. These models can significantly reduce training time and achieve better results without requiring large amounts of data. Notable pre-trained models such as BERT and GPT exhibit exceptional performance in natural language processing (NLP) and serve as milestones in the field of artificial intelligence. Due to their complex pre-training objectives and large parameter count, pre-trained models can effectively capture knowledge from vast amounts of labeled and unlabeled data. By storing knowledge in numerous parameters and fine-tuning specific tasks, the rich knowledge encoded in these parameters can benefit various downstream tasks, as demonstrated by experimental validation and empirical analysis. Pre-training mechanisms enable models to learn generic linguistic expressions by leveraging substantial volumes of unlabeled text. Pre-trained models can be adapted to downstream tasks by adding one or two specific layers, providing a good initialization and preventing the need to train downstream models from scratch. This approach improves performance on small datasets, reducing the requirement for a large number of labeled instances. As deep learning models with many parameters tend to overfit on small datasets, pre-training serves as a form of regularization by providing a good initialization and avoiding overfitting.

2.2. Self-attention mechanism

The self-attention mechanism constitutes a crucial component of pre-trained models commonly employed for NLP tasks, such as the Transformer. This mechanism determines the relationships between words by focusing on the input data's positions, resulting in more efficient text representations. The self-attention mechanism encodes the contextual information of an entire text in each word's semantic representation. Its advantages include capturing long-distance dependencies in text and effectively handling variable-length sequences. Owing to its outstanding performance in NLP, the self-attention mechanism has become a popular research topic in the field.

In our implementation, we utilize a multi-head self-attention mechanism. This approach replicates a single attention head into multiple ones, applying them to different data positions separately to obtain more semantic information. Each head independently learns distinct contextual information, enhancing the model's generalization capabilities. This method is widely used in Transformer models and has achieved satisfactory practical results.

2.3. Word embedding

Traditional machine learning methods often struggle to process textual data directly, necessitating appropriate techniques to convert text data into numerical data, which introduces the concept of word embedding. Word embedding encompasses language modeling and representation learning techniques in NLP, serving as an early pre-training technique. It refers to embedding a high-dimensional space, with dimensions equal to the total number of words, into a continuous vector space of much lower dimensionality. In this space, each word or phrase is mapped to a vector of real numbers, representing a specific concept through a distributed representation.

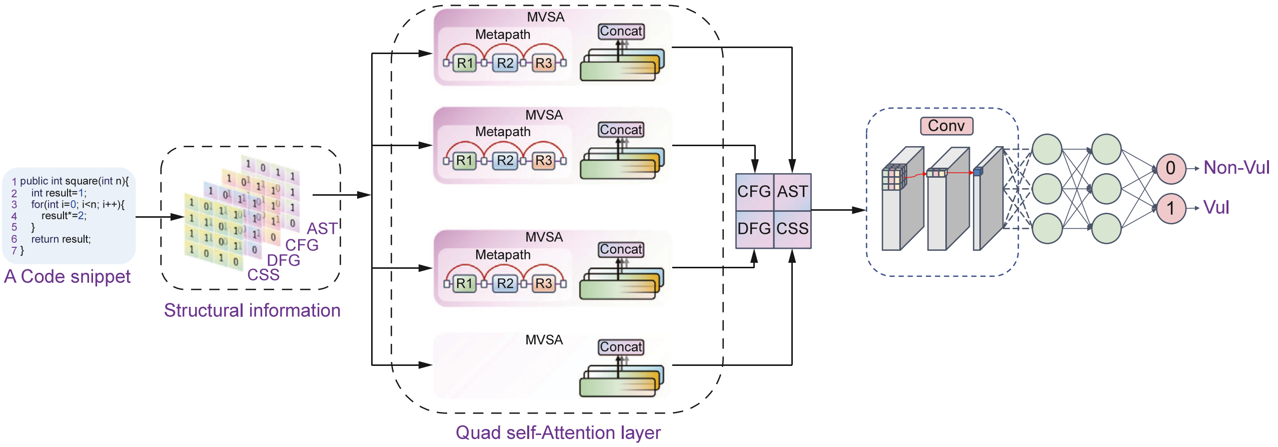
2.3.1. Bag-of-words

In information retrieval, the bag-of-words model assumes that a text is a collection of words or word combinations, disregarding word order, grammar, and syntax. The model considers the occurrence of each word in a text to be independent of the occurrences of other words. Common applications include one-hot encoding and N-gram techniques. Although the model is easy to understand and implement, it has several drawbacks. Careful vocabulary design is crucial, particularly to manage size and avoid sparse context representations. Disregarding word order neglects the context and meaning of words in a document.

2.3.2. Context-independent with machine learning

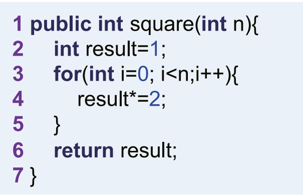
Context-independent machine learning approaches do not consider contextual information in the learning process. They assume input data is independent of other information, focusing solely on the current input during model training. These approaches are commonly used for text classification, with popular models including word2vec, fastText, and glove.

[3] https://github.com/WJ-8/JFinder



```
public int square(int n){
  int result=1;
  for(int i=0; i<n; i++){
    result*=2;
  }
  return result;
}
```

Fig. 1. Illustration of the JFinder model.



```
public int square(int n){
    int result=1;
    for(int i=0; i<n;i++){
        result*=2;
    }
    return result;
}
```

Fig. 2. A code example.

2.3.3. Context-dependent and transformer-based

Context-dependent and transformer-based methods represent context-sensitive approaches in which the same word is represented differently depending on the context. These methods obtain contextual information by using a transformer model to compute the relationships of the input data. Transformer models incorporate attention mechanisms that learn long-range dependencies, making them better suited for natural language processing tasks.

3. Approach

In this section, we detail the design of JFinder, illustrated in Fig. 1. The model workflow is as follows: we input a code snippet and parse it to obtain the AST, CFG, DFG, and CSS matrices. These matrices are then fed into the quad self-attentive layer and merged into a single matrix. Finally, a convolutional neural network and a multilayer perceptron predict whether the input code snippet is vulnerable or not.

3.1. Structural information

3.1.1. Abstract syntax tree

AST is a tree representation of the abstract syntactic structure of code written in a formal language. Each node of the tree denotes a construct occurring in the code, while each edge represents the inclusion relationship between the parent node and child nodes. In the tree, every sentence links to its tokens. Specifically, a sentence forms a concatenation graph with its tokens. For example, in Fig. 3, “int result = 1” links “int”, “result”, “=”, and “1”.

3.1.2. Control flow graph

The CFG is a graph-based representation of all pathways that a program may take during its execution. In other words, it illustrates how the program runs under various settings. All nodes belong to the branch set, which includes `switch`, `if`, `for`, and `while` statements. Each directed edge represents the program’s jumps between neighboring nodes and must follow a specific jump direction. The CFG has a unique entry and exit point. The program starts at the entry point and ends at the exit point. In Fig. 3, the `public int square(int n)` block is the entry point for the program shown in Fig. 2, and the `return result` block is the exit point. When the program reaches the `for(int i=0; i<n; i++)` block, the computer determines whether the value of the `i` variable is less than the value of the `n` variable. If the value of `i` is less than the value of `n`, the `for` block is executed; otherwise, the `return` block is executed. Thus, the `for` block is linked to both the `result*=2` block and the `return result` block.

3.1.3. Data flow graph

DFG is a graph that depicts the data dependencies between various operations, i.e., it records all variable creation and modification. In DFG, each node represents the creation or modification of variables, and each directed edge represents variables that have been modified. For example, in Fig. 3, the variable `result` is initialized in the `int result=1` block and is modified in the `return*=2` block. Therefore, we connect two nodes from `int result=1` to `result*=2` with a directed edge.

3.2. MetaPath

A MetaPath is a series of object-type relations that define a new composite relation between its initial and terminating type. It is represented as a path in the form of $\theta = A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_l} A_{l+1}$, where A_i is a state and R_i is a composite relation between A_i and A_{i+1} [2]. It defines a composite relation $R = R_1 \circ R_2 \circ \cdots \circ R_l$ between type A_1 and A_{l+1}, and $\circ$ denotes the composition operator on relations. The length of a MetaPath depends on the number of relations in different contexts. Predefining all potential MetaPaths of any length based on all conceivable node and edge types is challenging due to the exponential expansion of MetaPaths, increased data sparsity, and decreased training accuracy. A length-N MetaPath can be decomposed into (N-1) length-2 MetaPaths. We focus on length-2 MetaPaths [3,4] through reflective connections between adjacent nodes to extract multiple MetaPaths, i.e., we add a reverse directed edge for a pair of nodes with a directed edge. AST, CFG, and DFG are mostly tree-like, with very few back-edges. Adding the “back” relations improves the completeness of the extracted MetaPaths and enhances the connectivity of the graph to reduce overfitting.

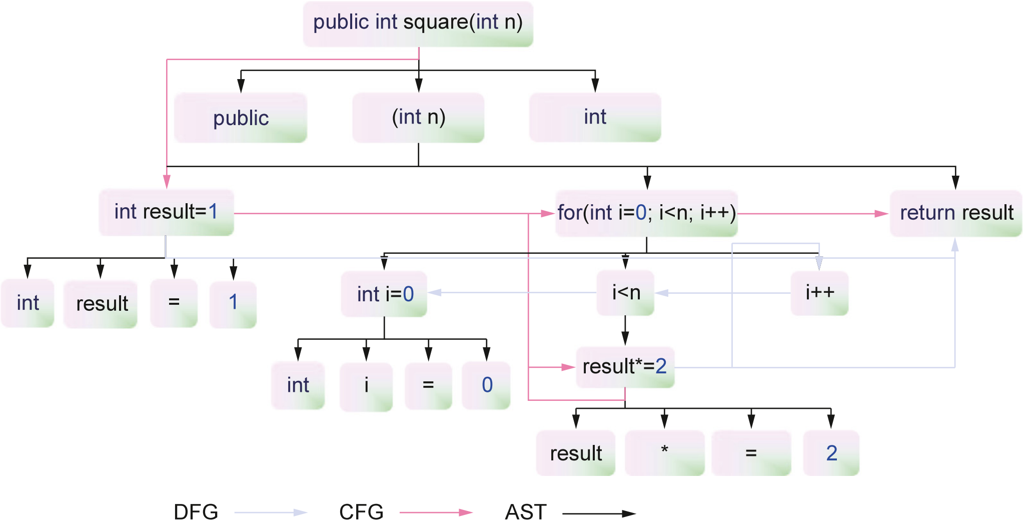


Fig. 3. Structural information.

3.3. Code snippet sequence

CSS is a novel program language encoder that represents the semantic information of a code snippet by encoding code snippets into feature information matrices. The key to CSS is using a pre-trained program language model. Compared to traditional encoding methods, pre-trained models achieve satisfactory results without training, while significantly reducing the computing power requirements. A program language pre-training model can obtain more appropriate features than a natural language pre-training model. We calculate CSS as follows:

$$C_i = model(x_i) \tag{1}$$

where x_i is an input code snippet, C_i is a representation of a code snippet and `model` represents a pre-trained program language model.

3.4. Multi-view self-attention encoder

After obtaining the structural information (AST, CFG, and DFG) and code semantic representation, we need to merge these representations and focus on location-specific information to extract more features. To achieve this, we design a multi-view self-attention encoder based on the multi-head attention mechanism (see Fig. 4). The input of MVSA consists of three matrices, Q, K, and V, which represent query, key, and value, respectively. Due to the self-attention mechanism, Q, K, and V are the same. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values. Single-head attention is calculated as shown in Eq. (2).

$$h_i = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) \tag{2}$$

where Q_i, K_i, and V_i denote the ith submatrix of Q, K, and V, respectively. d_k refers to the dimension of K. We concatenate all h_i from the scaled dot-product attention layer, calculated as shown in Eq. (3).

$$G = Concat(h_i, \dots, h_n)W^o \tag{3}$$

where W^o is a weight matrix that is trained jointly with the model, and n is a user-defined parameter. We concatenate all h_i as H and multiply it with W^o. The resulting G matrix captures information from all h_i. Finally, we assemble four MVSAs into a quadruple self-attention layer, calculated as follows:

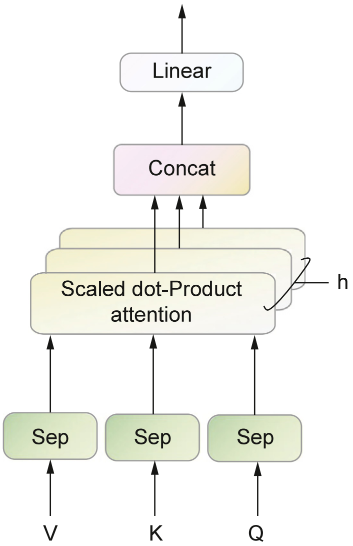


Fig. 4. Illustration of MVSA.

$$Q = Layer(AST, DFG, CFG, CSS) \tag{4}$$

where Q is a single matrix fusing of four matrices, containing both structural and semantic information.

4. Implementation

In this section, we provide details on the implementation of JFinder, as illustrated in Fig. 1, including data preparation and module implementation.

4.1. Generating data

4.1.1. Generating AST

We employed JavaParser,[4] an open-source tool for analyzing Java code, to construct abstract syntax trees (ASTs). Using the

[4] http://javaparser.org/

`parse` module, we generated an AST for a given code snippet and outputted a DOT file containing edge and node information. For instance, `n0 -> n1` denotes an edge from the 0th node to the 1st node. Based on this information, we constructed the AST adjacency matrix.

4.1.2. Generating CFG and DFG

We extracted C++ code using tree-sitter-c,[5] a parser-generating tool and incremental parsing library that produces concrete syntax trees for source files and efficiently updates the syntax trees as the source files change. We then created a Node class to store the current node's header nodes, end node, and next node, enabling straightforward traversal of all nodes and their associated nodes. Next, we used lists to store related nodes, added directed edges between them based on conditional expressions, and created the control flow graph adjacency matrix using these directed edges. For example, for node 4 with two child nodes 5 and 6, we set matrix entries $M(4,5)=1$ and $M(4,6)=1$.

The data flow graph adjacency matrix generation process is similar to that of the CFG adjacency matrix. The only difference is that we added directed edges based on data flow instead of conditional expressions.

4.1.3. Generating CSS

We employed the HuggingFace transformer library [5], which includes a framework of pre-trained models. We loaded the UniXcoder model [1] based on this framework, a unified cross-modal pre-trained model for programming languages that supports both code-related understanding and generation tasks. Before generating code snippet embeddings, we tokenized code snippets into token sequences. We used the UniXcoder tokenizer to process our datasets. However, the tokenizer's performance was suboptimal, as it divided `LF_NORMAL` into `LF`, `_`, and `NORMAL`. In our experiments, we found that incorrect code tokenization led to a significant decrease in our model's accuracy. To address this issue, we expanded the UniXcoder's vocabulary by traversing our datasets and recording words not present in the UniXcoder's vocabulary as a special word list, which we then added to the UniXcoder. For each source code snippet, the UniXcoder outputted a code semantic snippet embedding matrix containing its semantic information. We should note that the source code length cannot exceed 512 tokens, and any source code exceeding this length must be truncated.

4.2. Module implementation

4.2.1. Metapath

As shown in Fig. 1 and discussed in Section 3.2, we needed to add reversed edges for each pair of nodes connected by a directed edge. To customize our metapaths, we wrote a Python program to add metapaths to the AST, CFG, and DFG adjacency matrices, as shown below:

Algorithm 1: Metapath module

```
Input: Matrix $x_i \in AST, CFG, DFG$
Output: Matrix $y_i$
for $i=0$ $i< x_i.1st\text{-}dimention$ $i{+}{+}$ do
    for $q=0$ $q< x_i.2nd\text{-}dimention$ $q{+}{+}$ do
        if $x_i(i,q)=1$ then
            $x_i(q,i) \leftarrow 1$
        end
    end
end
```

4.2.2. Multi-view self-attention encoder

We implemented the MVSA using Keras [6] based on TensorFlow [7], an advanced neural network library for building and training deep learning models that offers an easy-to-use high-level interface for constructing, training, and evaluating deep learning models. We created a custom layer containing four MVSAs, named Quad Self-Attention Layer. The outputs of the Quad Self-Attention Layer were fed into a convolutional layer and a fully connected layer, which predicted whether a code snippet was vulnerable. Our model employed cross-entropy as the loss function, calculated as follows:

$$\mathcal{L}_{CE}=\sum_{i=0}^{N} y_i \log p_i + (1-y_i)\log(1-p_i), \tag{5}$$

where y_i represents the true label and p_i is the probability of the label predicted by the model.

5. Evaluation

5.1. Experimental setup

In this section, we describe the primary performance indicators for evaluating the model, the datasets, and the experimental environment.

5.1.1. Performance metrics

We employ F1 scores and accuracy to assess our model. These two metrics are widely used for evaluating model performance. We provide a brief explanation of these two metrics.

Accuracy: Accuracy is the ratio of the number of samples correctly predicted by the model to the total number of samples.

Precision: Precision is the ratio of the number of files correctly classified as vulnerable to the number of files classified as vulnerable.

Recall: Recall is the ratio of the number of files correctly classified as vulnerable to the total number of genuinely vulnerable files.

F1 Scores: F1 scores are the harmonic mean of precision and recall, calculated as follows:

$$2\times\frac{Recall\times Precision}{Recall+Precision} \tag{6}$$

5.1.2. Dataset

We use thirteen datasets to evaluate our model, including CWE datasets and the PROMISE dataset.[6] We selected CWE datasets from the NIST Software Assurance Reference Dataset[7] which is the software certification and evaluation division of the National Institute of Standards and Technology (NIST). This division is responsible for researching and developing software security assessment and certification standards. We chose six CWE datasets from SARD (shown in Table 1), including CWE 15, CWE 23, CWE 36, CWE 89, CWE 259, and CWE 606. These datasets consist of simplified vulnerability source codes derived from real software but have been artificially modified and patched. The PROMISE Repository is a publicly available repository specializing in software engineering research datasets. We selected the following Java project datasets: Camel, jEdit, Lucene, POI, Synapse, Xalan, and Xerces, which contain real software vulnerabilities without manual modification (shown in Table 2).

[5] https://tree-sitter.github.io/tree-sitter/

[6] http://openscience.us/repo/defect/

[7] https://samate.nist.gov

Table 1
CWE dataset information.

Project	Training set	Validation set	Test set	Total	Vul	Non-vul
CWE259	218	27	28	273	111	162
CWE606	775	97	97	969	333	636
CWE36	1046	131	131	1308	660	648
CWE15	1046	131	131	1308	660	648
CWE23	1046	131	131	1308	660	648
CWE89	3876	484	485	4845	1665	3180

Table 2
PROMISE dataset information.

Project	Training set	Validation set	Test set	Total	Vul	Non-vul
camel	2156	269	270	2695	562	2133
jedit	1309	164	164	1637	277	1360
lucene	600	75	75	750	437	313
poi	1086	136	136	1358	706	652
synapse	494	62	62	618	157	461
xalan	1831	229	229	2289	893	1396
xerces	835	104	105	1044	214	830

5.1.3. Environment configuration

Our experiment's hardware configuration was executed on a multi-core computing server featuring a 16-core 2.10 GHz Intel Xeon CPU and an NVIDIA 3090 GPU. The server has 256 GB of RAM and 24 GB of VRAM. The software configuration includes TensorFlow v2.7.0 and Keras v2.7.0 running on Windows 10. For MVSA, we set the head number to 4 for CSS, CFG, DFG, and AST. For the convolutional and fully connected layers, we use the Adam optimizer with a learning rate of 1e−5 and a batch size of 16. The overall training process takes approximately 5 min for each dataset. The final trained model has over 4,000,000 hyperparameters.

5.2. Baseline methods

5.2.1. Traditional [8]

The traditional method employs a Logistic Regression classifier based on 20 features.

5.2.2. DBN [9]

DBN utilizes a Deep Belief Network on source code to extract semantic features for defect prediction.

5.2.3. DBN+ [9]

DBN+, an improved version of DBN, combines semantic features with traditional features.

5.2.4. CNN [10]

CNN treats source codes as natural languages, with Word2Vec used for embedding initialization. It employs a CNN to extract features from source codes.

5.2.5. Defect Prediction via Convolutional Neural Network (DP-CNN) [9]

DP-CNN uses a CNN for automated feature generation from source code while preserving semantic and structural information. It employs word embedding and combines CNN-learned features with traditional hand-crafted features to further improve defect prediction.

5.2.6. Improved CNN [11]

The improved CNN model can learn semantic representations from source-code ASTs. It enhances global pattern capture ability and improves the model for better generalization.

5.2.7. Achilles [12]

Achilles is a Java source code security vulnerability detection tool based on LSTM RNN models. It can be trained using vulnerability source code datasets, analyze Java programs, and predict security vulnerabilities at the method level.

5.2.8. Intelligent Sentence-level Vulnerability Self-Detection Framework (ISVSF) [13]

ISVSF considers the syntax characteristics of Java and adopts sentence-level method representation and pattern exploration.

5.3. Experimental results

We evaluate our model on thirteen datasets shown in Table 1 and Table 2. JFinder outperforms all baselines in both conventional and highly complex datasets, demonstrating that our model represents the state of the art in Java vulnerability detection. According to the experimental results (shown in Table 4 and Table 3), we summarize the following findings:

- **Code structural information improves vulnerability identification performance.** Compared to Achilles and ISVSF, which incorporate ASTs without other structural information, our model's accuracy is more than 5% higher (see Fig. 7). Additionally, ISVSF outperforms Achilles in some datasets due to its inclusion of ASTs.
- **Pre-trained code semantic models enhance the ability of models to learn code representation semantics.** None of the baseline methods use a pre-training mechanism, resulting in their poor performance on real datasets. JFinder outperforms them by up to 35% in F1 scores (see Fig. 6).
- **Quadruple self-attention layer extracts code vulnerability patterns, and aggregates structural information and semantic representation of source code.** Comparing CNN, DP-CNN, and improved CNN, JFinder's F1 scores are more than 25% higher, indicating the positive effect of the quadruple self-attention layer.
- **JFinder performs better on highly complex datasets compared to conventional datasets.** Although JFinder surpasses all baseline methods in evaluations with commonly used datasets, its distinctiveness is not readily apparent. This is primarily due to the robust nature of the baseline methods, as well as researchers' adeptness in identifying elementary vulnerabilities. To underscore JFinder's superior performance, we employed a real-world dataset, which revealed JFinder to be 25% more effective in terms of F1 scores than the baseline method, signifying industry-ready performance standards. Notwithstanding, we discerned notable disparities in the model's performance across different datasets. The CWE dataset, being artificially created, comprises vulnerabilities that are relatively simplistic and easy to detect. Conversely, the PROMISE dataset, derived from open-source projects, encompasses more concealed and challenging-to-detect vulnerabilities. Yet, in the face of such complexity, our model demonstrates commendable performance.
- **JFinder achieves satisfactory performance with a short training time.** As seen in Fig. 5, accuracy and F1 scores increase dramatically in a short period of time. The model is trained in less than five minutes.

5.4. Ablation study

We conducted an ablation study to explore the impact of various components of our model on its performance. We utilized the PROMISE datasets to test the performance of the model components.

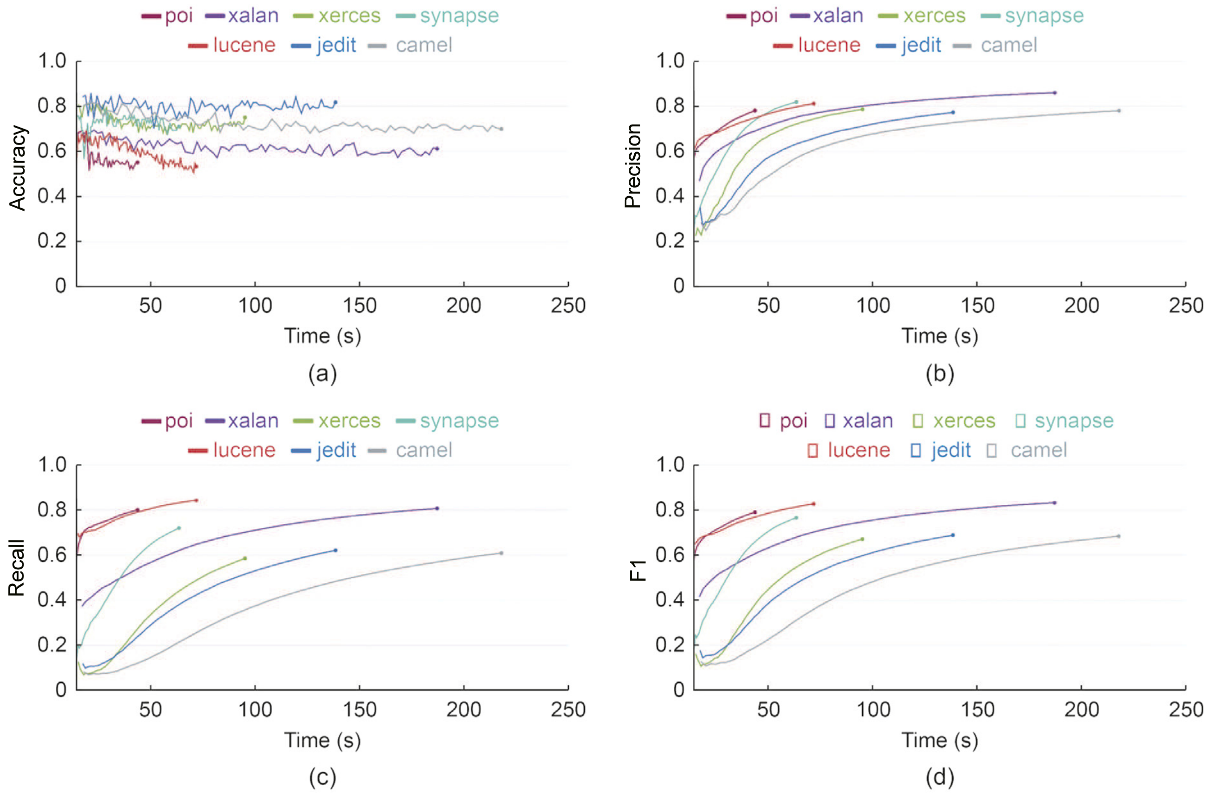


Fig. 5. Training information. (a) Validation Accuracy. (b) Validation Precision. (c) Validation Recall. (d) Validation F1.

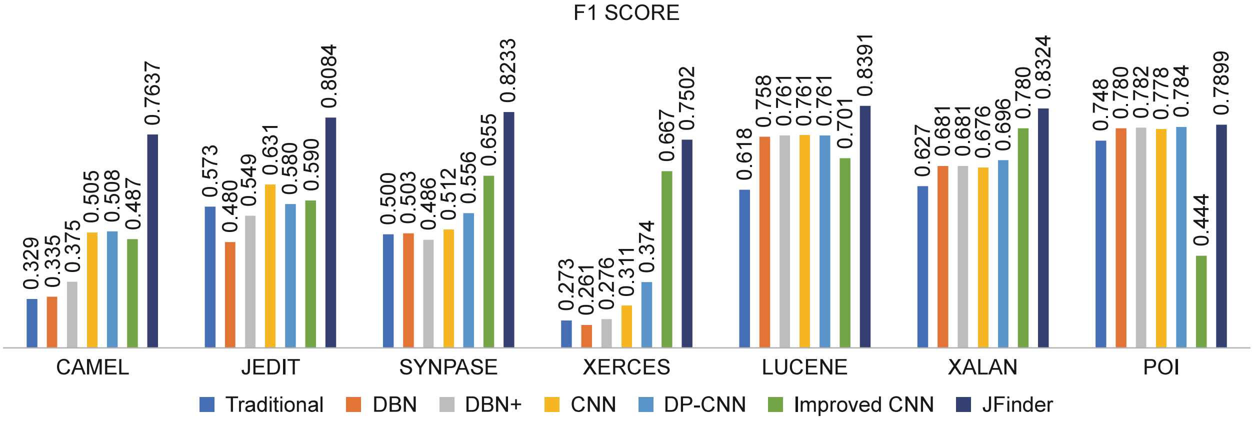


Fig. 6. F1 score on promise.

Table 3
Experimental results on PROMISE.

Method	camel	jEdit	synapse	xerces	lucene	xalan	poi
	F1	F1	F1	F1	F1	F1	F1
Traditional [8]	0.329	0.573	0.500	0.273	0.618	0.627	0.748
DBN [9]	0.335	0.480	0.503	0.261	0.758	0.681	0.780
DBN+ [14]	0.375	0.549	0.486	0.276	0.761	0.681	0.782
CNN [10]	0.505	0.631	0.512	0.311	0.761	0.676	0.778
DP-CNN [14]	0.508	0.580	0.556	0.374	0.761	0.696	0.784
Improved CNN [11]	0.487	0.590	0.655	0.667	0.701	0.780	0.444
JFinder	0.7637	0.8084	0.8233	0.7502	0.8391	0.8324	0.7899

Table 4
Experimental results on CWE.

Method	CWE259	CWE606	CWE36	CWE15	CWE23	CWE89
	ACC	ACC	ACC	ACC	ACC	ACC
Achilles [12]	0.925	0.943	0.818	0.929	0.894	0.934
ISVSF [13]	0.8732	0.9421	0.9321	0.9289	0.9305	0.9523
JFinder	0.9643	0.9691	0.9771	0.9466	0.9542	0.9610

- **Structural information**

  In the ablation experiment, we removed one of the structural information matrices, as shown in Table 5. According to the table, removing any of the structural information matrices resulted in a decrease in F1 scores, indicating that structural information provides a significant number of features to the model. Upon analyzing the results further, we found that CFG and DFG are more critical than AST. When either CFG or DFG was removed, the F1 scores dropped more substantially. However, for the `poi` dataset, they played similar roles. Thus, we infer that CFG and DFG provide

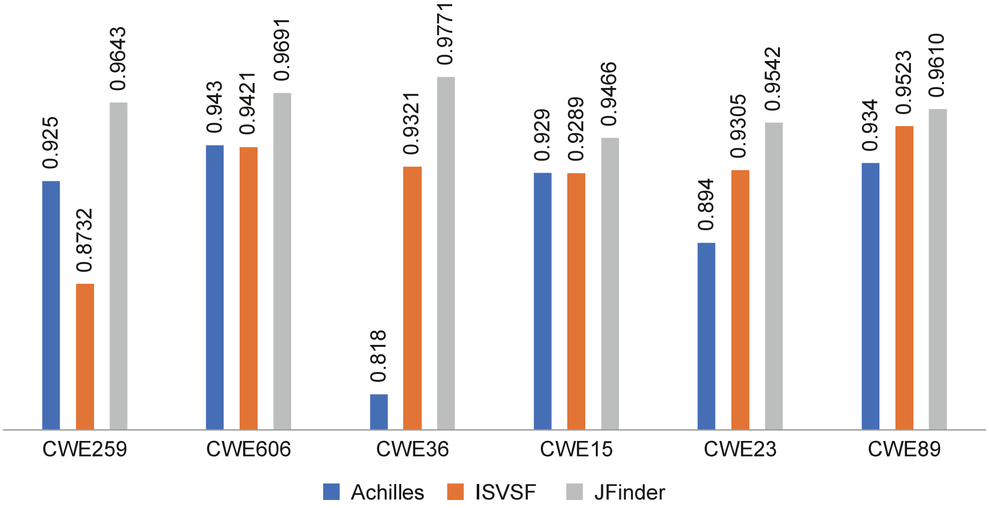


Fig. 7. Accuracy on CWE.

Table 5
Ablation study.

Method	camel	jEdit	synapse	xerces	lucene	xalan	poi
	F1	F1	F1	F1	F1	F1	F1
JFinder	0.7637	0.8084	0.8233	0.7502	0.8391	0.8324	0.7899
Jfinder w/o AST	0.7204	0.7904	0.8164	0.7443	0.8273	0.8085	0.7730
Jfinder w/o CFG	0.7343	0.7551	0.8199	0.6982	0.6292	0.7028	0.7603
JFinder w/o DFG	0.5963	0.7378	0.7763	0.7342	0.5814	0.7283	0.7764
JFinder w/o CSS	0.4895	0.5933	0.7111	0.5159	0.6882	0.6760	0.7166

the model with data flow and control flow information containing more vulnerability patterns. In summary, it is unwise to remove any of the graphs, even if they appear less important.

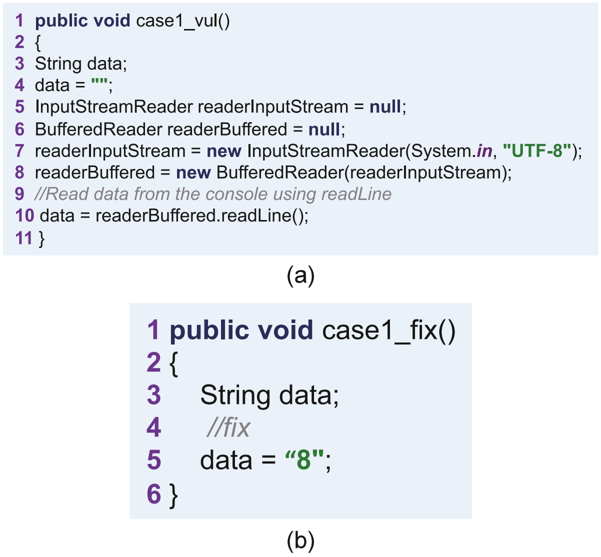
- **Semantic information and pre-trained model**
  Based on Table 5, we conclude that the semantic information of the source code is highly important, as it is the origin of most vulnerability features. Removing CSS resulted in a significant drop in F1 scores to 0.51. Evidently, CSS is the most critical information. Our novel quadruple self-attention layer design and pre-training mechanism play the most significant role. If this component is removed, the model becomes inoperative.

5.5. Case study

To further assess the robustness and intelligence of JFinder, we examined four representative vulnerability cases in the CWE dataset. Our model correctly identified each case. First, we analyzed the vulnerability specifications. Next, we manually patched the vulnerabilities and input them into our model to determine if it no longer flagged them, checking its ability to deeply understand the meaning of the vulnerabilities.

5.5.1. Case 1

In Case 1 (Fig. 8), the program did not check input and read data from the console using `readLine()`, potentially causing a denial of service or other consequences due to excessive looping. The data originated from a malicious source, as shown in Fig. 8(a) Line 10. To remediate the vulnerability (Fig. 8(b)), we replaced user-controlled data for loop conditions with a hardcoded string. Afterward, JFinder no longer flagged this source code, indicating its ability to learn the reason behind unchecked input for loop condition problems.



```
1 public void case1_vul()
2 {
3 String data;
4 data = "";
5 InputStreamReader readerInputStream = null;
6 BufferedReader readerBuffered = null;
7 readerInputStream = new InputStreamReader(System.in, "UTF-8");
8 readerBuffered = new BufferedReader(readerInputStream);
9 //Read data from the console using readLine
10 data = readerBuffered.readLine();
11 }
```

(a)

```
1 public void case1_fix()
2 {
3     String data;
4      //fix
5     data = “8";
6 }
```

(b)

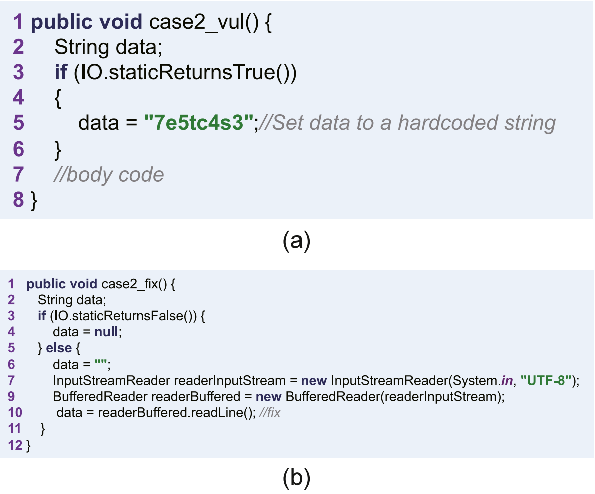
Fig. 8. Case 1. (a) Vulnerability. (b) Fix.

5.5.2. Case 2

In Case 2 (Fig. 9), the code snippet exhibited a vulnerability involving the use of a hard-coded password. A hard-coded password can lead to significant authentication failures that may be difficult for system administrators to detect and fix. In extreme cases, administrators may be forced to disable the product entirely. As seen in Fig. 9(a) Line 5, the program established a hard-coded password, suggesting that if such passwords are used, malicious users are likely to gain access through the account in question. To patch the vulnerability, we set `data` via external input, as shown in Fig. 9(b). After testing, JFinder no longer flagged the program as vulnerable.

5.5.3. Case 3

In Case 3 (Fig. 10), a vulnerability appeared in Line 7, as depicted in Fig. 10(a). In Line 8, the program read data from a properties file, leading to SQL injection. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as



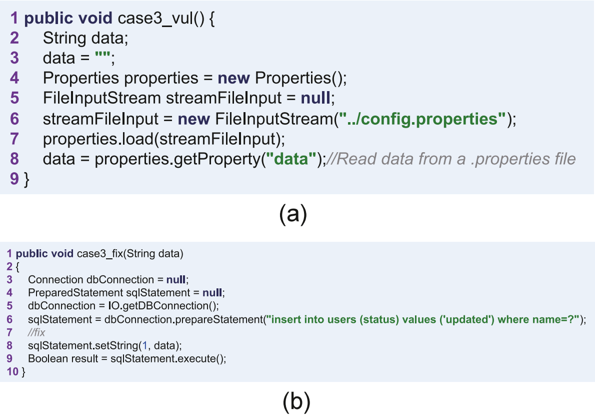
```
public void case2_vul() {
    String data;
    if (IO.staticReturnsTrue())
    {
        data = "7e5tc4s3";//Set data to a hardcoded string
    }
    //body code
}
```

(a)

```
public void case2_fix() {
   String data;
   if (IO.staticReturnsFalse()) {
       data = null;
   } else {
       data = "";
       InputStreamReader readerInputStream = new InputStreamReader(System.in, "UTF-8");
       BufferedReader readerBuffered = new BufferedReader(readerInputStream);
        data = readerBuffered.readLine(); //fix
    }
}
```

(b)

Fig. 9. Case 2. (a) Vulnerability. (b) Fix.



```
public void case3_vul() {
    String data;
    data = "";
    Properties properties = new Properties();
    FileInputStream streamFileInput = null;
    streamFileInput = new FileInputStream("../config.properties");
    properties.load(streamFileInput);
    data = properties.getProperty("data");//Read data from a .properties file
}
```

(a)

```
public void case3_fix(String data)
{
    Connection dbConnection = null;
    PreparedStatement sqlStatement = null;
    dbConnection = IO.getDBConnection();
    sqlStatement = dbConnection.prepareStatement("insert into users (status) values ('updated') where name=?");
    //fix
    sqlStatement.setString(1, data);
    Boolean result = sqlStatement.execute();
}
```

(b)

Fig. 10. Case 3. (a) Vulnerability. (b) Fix.

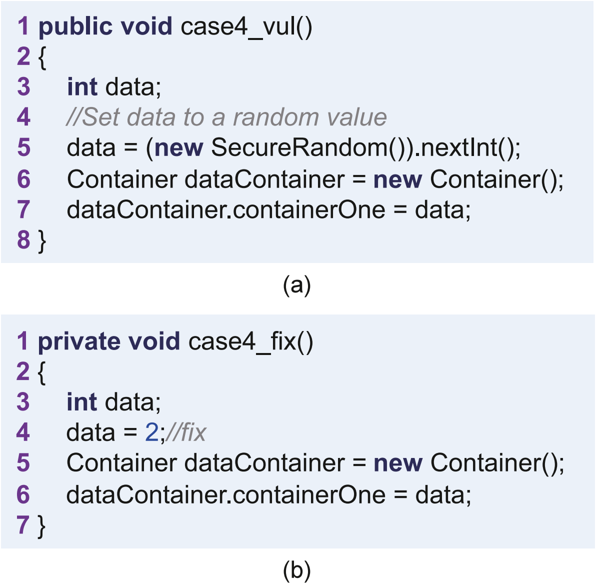
SQL rather than ordinary user data. This can be exploited to alter query logic, bypass security checks, or insert additional statements that modify the back-end database, potentially including the execution of system commands. To fix the vulnerability, we set `data` to be passed through the function instead of being read from the properties file. Afterward, JFinder considered the code snippet non-vulnerable.

5.5.4. Case 4

In Case 4 (Fig. 11), a vulnerability arose from allocating memory based on an untrusted, large size value. The program did not ensure that the size was within expected limits, allowing arbitrary amounts of memory to be allocated. As shown in Fig. 11(a) Line 5, the program set `data` to a random value. To address this issue, we used a hardcoded number that would not cause underflow, overflow, divide by zero, or loss-of-precision problems. After fixing it in Fig. 11(b), JFinder no longer deemed it a vulnerability.

6. Related works

The automation of software vulnerability identification is a topic of great interest for researchers, who continue to develop new techniques to detect vulnerabilities. Approaches range from traditional methods that manually establish vulnerability rules, to machine learning techniques that determine vulnerabilities based on features, and to deep learning that learns vulnerability patterns.



```
1 public void case4_vul()
2 {
3   int data;
4   //Set data to a random value
5   data = (new SecureRandom()).nextInt();
6   Container dataContainer = new Container();
7   dataContainer.containerOne = data;
8 }
```

(a)

```
1 private void case4_fix()
2 {
3   int data;
4   data = 2;//fix
5   Container dataContainer = new Container();
6   dataContainer.containerOne = data;
7 }
```

(b)

Fig. 11. Case 4. (a) Vulnerability. (b) Fix.

6.1. Traditional detection methods

Traditional detection methods rely on known vulnerability rules to detect vulnerabilities, using manual code reviews and automated code scanners. Kaur et al. introduced five static code analysis tools for vulnerability detection in C/C++ and Java [15]. Flawfinder [16] is designed to detect vulnerabilities in C/C++ source code, generating a list of vulnerabilities for the program sorted by risk level. RATS [17] is a security vulnerability auditing tool for C/C++ source code that can detect issues such as buffer overflows. SPOTBUGS [18] is a program that uses static analysis to identify bugs in Java code, checking for more than 400 bug patterns. PMD [19] is an open-source source analysis tool that employs rule sets to find common coding errors, irregular code, and potential vulnerabilities. Peguero et al. analyzed Electron application security, revealing potential front-to-back-end attack escalation. They proposed framework modifications and an IDE plugin for early vulnerability fixes. Their studies confirmed the effectiveness of the plugin, as applications ceased to be exploitable post-fix [20].

6.2. Machine learning-based methods

When integrating various types of conventional hand-crafted features, the challenge remains how to effectively combine these features. Machine learning-based methods can address this issue by performing simple classification on manually extracted features with better performance than traditional methods. Yang et al. proposed a deep learning model for just-in-time defect prediction [21], building a set of expressive features from a set of initial change features using a deep belief network algorithm and constructing a classifier based on the selected features. Chen et al. proposed a model [22] capable of identifying SQL injection vulnerabilities by processing HTTP request text data using word2vec and classifying processed samples with the SVM algorithm. Al-Yaseen et al. suggested a multi-level hybrid intrusion detection model that employs support vector machines and extreme learning machines to enhance the efficiency of detecting known and unknown attacks [23]. Ren et al. introduced DVCMA [24], a software vulnerability detection method based on clustering and model analysis that applies clustering techniques to mine patterns from vulnerability sequences. Peguero et al. analyzes cross-site request forgery vulnerabilities in several server-side JavaScript frameworks. Utilizing automated static analysis, the security efficacy of each is evaluated. Based on these insights, recommendations for more secure application development are provided [25].

6.3. Deep learning-based methods

Manual inspection of source code or manual extraction of features from the source code is time-consuming. Deep learning-based methods can automatically extract vulnerability patterns and classify them based on the input source code's features. Zhan et al. proposed ISVSF [13], an intelligent sentence-level vulnerability self-detection framework that considers Java syntax characteristics and adopts sentence-level method representation and pattern exploration. Malhotra et al. suggested an improved CNN [11], a modified CNN algorithm that combines CNN-based layers into one and then applies a concatenate algorithm under SVM. Saccente et al. introduced Project Achilles [12], a Java source code security vulnerability detection tool built upon LSTM RNN models. Lin et al. proposed VulEye [26], a graph neural network vulnerability detection approach for PHP applications that utilizes program dependence graphs as input and is trained with a graph neural network model containing three stack units. Zheng et al. presented CodeGeeX, a multilingual, 13 billion-parameter model surpassing peers in code generation and translation on HumanEval-X. The model enhances coding efficiency for 83.4% of users through developed extensions. In September 2022, all associated CodeGeeX resources were publicly released [27]. Raymond Li et al. presented StarCoder and StarCoderBase, Code LLMs with 15.5 B parameters and 8 K context length. Trained on The Stack's 1 trillion tokens, StarCoder, a fine-tuned StarCoderBase, outperforms multilingual Code LLMs and Python-specialized models [28].

7. Conclusions and future research

As modern software grows in size and complexity, ensuring stability has become a critical concern. In this paper, we introduced JFinder, a novel architecture for Java vulnerability identification based on quad self-attention and a pre-training mechanism. JFinder innovatively combines structural and semantic information through a proposed quad self-attention layer. Experimental results demonstrate that JFinder outperforms all baseline models, achieving a level of performance suitable for industrial use. JFinder's F1 scores are 25% higher than those of the baselines, and its ACC surpasses them by 5%. Furthermore, we conducted a case study to investigate whether the model truly understands the root causes of vulnerabilities. Looking forward, we see significant potential in exploring the use of larger language models for pre-training. The pre-training mechanism, an unsupervised learning process, enables the model to learn a wide range of syntactic and semantic patterns before fine-tuning on a specific task. The application of larger language models in this process could enhance our model's understanding of complex code structures and semantics, possibly leading to improved precision and recall in identifying vulnerabilities in Java code.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This study was supported by the National Key R&D Program of China (2019YFB2102600), the National Natural Science Foundation of China (62002067), the Guangzhou Youth Talent of Science (QT20220101174), and the Project of Philosophy and Social Science Planning of GuangDong (GD21YGL16).

References

[1] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, UniXcoder: Unified cross-modal pre-training for code representation, 2022, arXiv preprint arXiv:2203.03850.
[2] H.H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudenko, T.-N. Doan, L. Jiang, MANDO: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities, in: Proceedings of the 9th IEEE International Conference on Data Science and Advanced Analytics, DSAA '22, 2022.
[3] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, P.S. Yu, Heterogeneous graph attention network, in: The World Wide Web Conference, WWW '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450366748, 2019, pp. 2022–2032, http://dx.doi.org/10.1145/3308558.3313562.
[4] Y. Sun, J. Han, X. Yan, P.S. Yu, T. Wu, PathSim: Meta path-based top-K similarity search in heterogeneous information networks, Proc. VLDB Endow. 4 (11) (2020) 992–1003, http://dx.doi.org/10.14778/3402707.3402736.
[5] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T.L. Scao, S. Gugger, M. Drame, Q. Lhoest, A.M. Rush, Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Association for Computational Linguistics, 2020, pp. 38–45, Online. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.
[6] F. Chollet, et al., Keras, GitHub, 2015, https://github.com/fchollet/keras.
[7] M. Abadi, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation, {OSDI} 16, 2016, pp. 265–283.
[8] Z. He, F. Peters, T. Menzies, Y. Yang, Learning from open-source projects: An empirical study on defect prediction, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 45–54, http://dx.doi.org/10.1109/ESEM.2013.20.
[9] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: 2016 IEEE/ACM 38th International Conference on Software Engineering, ICSE, 2016, pp. 297–308, http://dx.doi.org/10.1145/2884781.2884804.
[10] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, Association for Computational Linguistics, Doha, Qatar, 2014, pp. 1746–1751, http://dx.doi.org/10.3115/v1/D14-1181, URL https://aclanthology.org/D14-1181.
[11] R. Malohtra, H.S. Yadav, An improved CNN-based architecture for within-project software defect prediction, in: V.S. Reddy, V.K. Prasad, J. Wang, K.T.V. Reddy (Eds.), Soft Computing and Signal Processing, Springer, Singapore, ISBN: 978-981-33-6912-2, 2021, pp. 335–349.
[12] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, Y. Xiong, Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop, ASEW, 2019, pp. 114–121, http://dx.doi.org/10.1109/ASEW.2019.00040.
[13] H. Zhang, Y. Bi, H. Guo, W. Sun, J. Li, ISVSF: Intelligent vulnerability detection against java via sentence-level pattern exploring, IEEE Syst. J. 16 (1) (2022) 1032–1043, http://dx.doi.org/10.1109/JSYST.2021.3072154.
[14] J. Li, P. He, J. Zhu, M.R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS, 2017, pp. 318–328, http://dx.doi.org/10.1109/QRS.2017.42.
[15] A. Kaur, R. Nayyar, A comparative study of static code analysis tools for vulnerability detection in C/C++ and JAVA source code, Procedia Comput. Sci. 171 (2020) 2023–2029, http://dx.doi.org/10.1016/j.procs.2020.04.217, URL https://www.sciencedirect.com/science/article/pii/S1877050920312023. Third International Conference on Computing and Network Communications (CoCoNet'19).
[16] Flawfinder, 2001, https://dwheeler.com/flawfinder/. (undefined 27/1/2023 17:32).
[17] RATS, 2016, https://code.google.com/archive/p/rough-auditing-tool-for-security/. (undefined 27/1/2023 18:47).
[18] SpotBugs, 2023, https://spotbugs.github.io/. (undefined 27/1/2023 18:51).

[19] PMD, 2015, https://pmd.github.io/. (undefined 27/1/2023 18:56).

[20] K. Peguero, X. Cheng, Electrolint and security of electron applications, High-Conf. Comput. 1 (2) (2021) 100032, http://dx.doi.org/10.1016/j.hcc.2021.100032, URL https://www.sciencedirect.com/science/article/pii/S2667295221000222.

[21] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun, Deep learning for just-in-time defect prediction, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 17–26, http://dx.doi.org/10.1109/QRS.2015.14.

[22] Z. Chen, M. Guo, L. zhou, Research on SQL injection detection technology based on SVM, MATEC Web Conf. 173 (2018) 01004, http://dx.doi.org/10.1051/matecconf/201817301004.

[23] W.L. Al-Yaseen, Z.A. Othman, M.Z.A. Nazri, Multi-level hybrid support vector machine and extreme learning machine based on modified K-means for intrusion detection system, Expert Syst. Appl. 67 (C) (2017) 296–303, http://dx.doi.org/10.1016/j.eswa.2016.09.041.

[24] J. Ren, B. Cai, H. He, C. Hu, A method for detecting software vulnerabilities based on clustering and model analyzing, J. Comput. Inf. Syst. 7 (2011).

[25] K. Peguero, X. Cheng, CSRF protection in JavaScript frameworks and the security of JavaScript applications, High-Conf. Comput. 1 (2) (2021) 100035, http://dx.doi.org/10.1016/j.hcc.2021.100035, URL https://www.sciencedirect.com/science/article/pii/S2667295221000258.

[26] C. Lin, Y. Xu, Y. Fang, Z. Liu, VulEye: A novel graph neural network vulnerability detection approach for PHP application, Appl. Sci. 13 (2) (2023) http://dx.doi.org/10.3390/app13020825, URL https://www.mdpi.com/2076-3417/13/2/825.

[27] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, J. Tang, CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X, 2023, arXiv:2303.17568.

[28] R. Li, L.B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T.Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L.K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S.S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C.J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C.M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, StarCoder: may the source be with you!, 2023, arXiv:2305.06161.