Journal of Computers Vol. 37 No. 5, October 2026, pp. 131-152
https://doi.org/10.63367/19911599202610370500 9

Implementation of an Effective Code Vulnerability Detection Model Using Deep Learning and Attention Mechanism

Er-Zhou Zhu[1], Le-Hao Qi[1], Jia-Qi Chang[1], and Zhu-Juan Ma[2*]

[1] School of Computer Science and Technology, Anhui University, Hefei 230601, China
ezzhu@ahu.edu.cn, dale_qi_777@163.com, cjqi1998@163.com

[2] School of Big Data and Artificial Intelligence, Anhui Xinhua University, Hefei 230088, China
zjmsjtu16@163.com

Received 26 July 2025; Revised 2 February 2026; Accepted 13 February 2026

Abstract. With increasing computational power, deep learning methods have been widely applied to software vulnerability detection. Compared to traditional machine learning, deep learning offers low computational overhead, high accuracy, and eliminates the need for manual feature engineering. However, many existing approaches lack sufficient utilization of code syntax and semantics, limiting performance. This study proposes VDDA (Vulnerability Detection based on Deep learning and Attention mechanisms), an effective model that combines deep learning with attention mechanisms. VDDA employs a Bidirectional Long Short-Term Memory (BLSTM) network to capture code features automatically. Source code is transformed into Code Property Graphs (CPG) using the Joern tool to retain rich syntactic and semantic information. To enhance feature representation, several strategies are applied: depth-first traversal-based CPG optimization, three-directional code slicing, code block-based slice organization, and separation of function and variable names during symbolization. These methods convert source code into vector representations suitable as input to the deep model. An attention mechanism is integrated with BLSTM to emphasize the most relevant parts of the input vectors for improved vulnerability detection. Experiments on two datasets of varying sizes demonstrate that VDDA achieves superior detection performance compared to existing methods, validating its effectiveness and generalizability in software vulnerability detection tasks.

Keywords: software security, vulnerability detection, deep learning, attention mechanism

1 Introduction

The ever-increasing scale of code writing and the division of work involved make it difficult to guarantee the quality of software. The existence and wide application of a large number of free software programs bring convenience to people's work and daily life, but also increasingly produce security problems and hidden dangers. Security-related defects and hidden dangers in the software development process are collectively referred to as security vulnerabilities. An attacker can exploit security vulnerabilities to carry out attacks such as data tampering and leakage, gaining control of the target platform, and destroying the functions of the target platform. Most current software attacks are launched by exploiting vulnerabilities in the software itself [1]. In 2024, a notable 768 CVE-identified vulnerabilities were reported to have been exploited in the wild. This marks a 20% increase when compared to the 639 CVEs reported in 2023 [2]. In the domain of program security, research focused on exploiting software flaws or vulnerabilities, whether prior to or subsequent to corresponding attacks, has grown increasingly significant [3].

With the development of artificial intelligence and the greater availability of computational power, many machine learning methods that cover scenarios such as static analysis, dynamic analysis, and combined dynamic and static analysis have been proposed to detect software vulnerabilities [4]. Traditional machine learning methods generally adopt feature engineering as the main approach for vulnerability detection. However, the accuracy of feature engineering systems is heavily dependent on the prior knowledge of features. In addition, extracting comprehensive features from different dimensions for high detection accuracy is time-consuming, and attackers can bypass any established detection systems by mastering artificial features [5]. For example, the SmartMixModel [6] vulnerability detection model computes the context based on the abstract syntax tree

* Corresponding Author

(AST) of the source code. By utilizing random forest (RF) and support vector machine (SVM) as the underlying classifiers, this model can effectively detect known vulnerabilities. Owing to the faultiness of the feature extraction process, the detection accuracy and types of recognized vulnerabilities of this model need to be improved and extended.

Due to their strong learning ability and their ability to extract vulnerability features without expert knowledge, deep learning methods have been widely studied and used in software vulnerability detection [7]. Moreover, software vulnerability detection methods that are based on deep learning frameworks are more accurate and efficient than traditional machine learning frameworks. Deep learning vulnerability detection frameworks generally convert source code into vectors that can accurately express the characteristics of vulnerabilities. For example, the SySeVR [8] deep learning vulnerability detection framework converts source code into token sequences. The vulnerability detection performance of this method was significantly better than that of traditional machine learning methods. However, much of the syntax and semantic information in this framework is lost after the token sequence is converted into vectors.

Syntax and semantic information are important factors that have a significant impact on the performance of deep learning vulnerability detection methods. In real works on detecting vulnerabilities in software, the performance of many existing deep learning methods is degraded by the deficiency of syntax and semantic information. Studies have shown that data dependence and control dependence-induced semantic information can reduce the false-negative rate by 30.4% on average [8].

To retain richer syntactic and semantic information, many deep learning-based software vulnerability detection methods convert source code into graph-structured representations [9]. Traditionally, to comprehensively characterize program behavior, researchers construct various program graphs, such as Control Dependence Graphs (CDG), Data Dependence Graphs (DDG), Control Flow Graphs (CFG), and Program Dependence Graphs (PDG). By integrating the information contained in these graphs, the Code Property Graph (CPG) has become an important and highly expressive code representation in the field of program analysis [10]. However, while preserving comprehensive syntax and semantics, this approach inevitably leads to a high degree of CPG structural complexity. Existing CPG-based vulnerability detection methods, such as FUNDED [11] and VulDetector [12], typically construct complete or near-complete CPGs to uniformly retain control and data dependencies to maximize semantic coverage. As program complexity increases, this design causes the rapid expansion of the graph scale, a problem that further introduces significant computational overhead during graph construction, traversal, and model inference. For instance, FUNDED relies on dependency propagation mechanisms over large-scale graph structures. When analyzing complex functions or large programs, its training and inference costs increase significantly. Similarly, while VulDetector encodes comprehensive dependency relationships, it inevitably introduces a large amount of dependency information unrelated to vulnerabilities, adversely affecting detection efficiency and accuracy in large-scale scenarios.

More critically, many dependencies preserved in a complete CPG have low correlation with vulnerability triggers. Methods like FUNDED and VulDetector treat different types of control and data dependencies equally during modeling, which easily introduces structural redundancy and thereby dilutes the critical semantic information directly related to vulnerabilities [13]. In actual software systems, vulnerabilities are often triggered by specific operations (such as API calls, arithmetic expressions, or pointer operations) whose semantic context typically exhibits distinct locality features. Consequently, learning directly on complete CPGs may obscure these localities, further degrading vulnerability detection efficiency.

In this study, we propose VDDA (Vulnerability Detection based on Deep learning and Attention mechanisms), an effective software vulnerability detection model leveraging deep learning and attention mechanisms. In VDDA, we employ the Bidirectional Long Short-Term Memory (BLSTM) network as the core deep learning model to automatically extract features and alleviate the reliance on manual feature engineering inherent in traditional machine learning techniques. To enhance the quality of code representation, we adopt the Joern analysis tool [14] to generate the initial CPG from source code. Subsequently, a depth-first traversal-based CPG optimization algorithm is designed to simplify the graph structure prior to vectorization. Furthermore, we introduce several key improvements in the preprocessing stage, specifically, three-direction code slicing, slice organization via code blocks, and the separation of function names from variable names during code symbolization, to effectively convert source code into vector inputs suitable for the deep model. Recognizing that different parts of the code vector contribute unequally to vulnerability detection, we incorporate an attention mechanism to dynamically weight these contributions. This allows the model to focus on critical vulnerability indicators, thereby further boosting detection performance. The contributions made in the VDDA are summarized as follows:

(1) *It offers an effective method for converting source code into vectors*. Because vectors are used as input, an effective vectorization method that can simplify the learning process without sacrificing performance is an important factor in improving the detection efficiency of deep learning vulnerability models. In the VDDA, several improvements, including three-direction code slicing, slice organization with code blocks, and separating function names from variable names in code symbolization, are made to effectively convert source code into vectors for the underlying deep model.

(2) *It contains a CPG optimization algorithm*. The CPG contains comprehensive syntax and semantic information that can be obtained by a combination of traditional graphs, such as AST, CDG, DDG, CFG, and PDG. However, because it is more complex than the other graphs, it must be optimized before it is vectorized. To improve the vulnerability detection efficiency, the depth-first traversal-based CPG optimization algorithm is proposed to remove the redundant control flow edges and control dependency edges dynamically.

(3) *It contains a deep learning and attention mechanism combined architecture for vulnerability detection*. Because most vulnerabilities are not caused by just one line of code but are closely related to the code context, BLSTM is used to detect software vulnerabilities. In a code block, the importance of the different slices to vulnerability detection is not the same. Meanwhile, the importance of code statements and tokens within a slice also differs. If all these components are treated equally, the vulnerability detection performance degrades. In this study, the attention mechanism was incorporated into a deep learning framework to improve the vulnerability detection performance.

This study is a continuation of the work presented in [15]. On the basis of that paper, we have made tremendous improvements. The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 details the VDDA model and Section 4 presents the experimental results. Finally, Section 5 concludes the paper and outlines future work.

2 Related Work

The primary purpose of software vulnerability detection is to identify and mitigate security flaws within source code prior to exploitation [16]. Given that vulnerabilities introduced during software development can lead to severe consequences, such as data leakage, privilege escalation, and system compromise, it has long been a fundamental research topic in program security.

Existing techniques can be broadly categorized by their analysis technology and intermediate representations. Representative approaches include code similarity-based, symbolic execution-based, rule-based, and machine learning-based detection [17]. Among these, machine learning–based methods have gained increasing prominence due to their ability to automatically learn vulnerability patterns from large-scale code corpora. In the following subsections, we review related work by progressively introducing traditional machine learning, deep learning with textual representations, graph-based deep learning, and recent applications of large language models.

2.1 Traditional Machine Learning–based Vulnerability Detection

With the development of artificial intelligence and the increasing availability of computational power, machine learning methods have been widely studied and used to detect software vulnerabilities [18]. Traditional machine learning methods rely on manually engineered features extracted from source code or intermediate representations. Early studies typically adopt statistical or rule-based feature engineering, followed by classifiers such as SVMs and RFs, to identify vulnerable patterns [19]. These approaches can effectively detect known vulnerability types when domain-specific features are meticulously crafted.

However, their effectiveness is heavily dependent on the quality and completeness of these handcrafted features. This process is not only time-consuming and knowledge-intensive but also renders the methods difficult to scale to large and diverse codebases. Moreover, manual features are often brittle. Once attackers understand the detection logic, they can easily bypass it, which will result in limited generalization in real-world scenarios [20].

2.2 Deep Learning with Textual Code Representations

To alleviate the reliance on manual feature engineering, deep learning–based vulnerability detection has been widely investigated. Most approaches represent source code as token sequences and employ neural architectures to automatically learn patterns. SySeVR [8], a representative deep learning-based vulnerability detection framework, first extracts vulnerability-relevant syntactic cues, converts them into semantic sequences, and subsequently feeds the vectorized intermediate code representations to a neural network. However, by treating source code purely as a token sequence, this approach inevitably loses fine-grained syntactic and semantic information during the vectorization process. MDVul [13] targets code structures with complex dependencies. By integrating semantic information and multi-path fusion techniques, it enhances both the accuracy and efficiency of vulnerability identification. This approach effectively addresses the limitations of traditional methods in handling intricate code dependencies through improved semantic representation and dependency path modeling.

VulBERTa [21] is a deep learning model specifically designed to learn robust representations of C/C++ code from large-scale, diverse codebases. Leveraging a language-aware tokenization pipeline, it parses and tokenizes source code while preserving essential syntactic and semantic features. Chakraborty et al. [22] evaluated four state-of-the-art deep learning–based vulnerability detectors and observed a significant performance drop—average precision fell by over 50% compared to benchmark reports. Key factors included limited model expressiveness, spurious feature learning, data duplication, and class imbalance. To address these issues, they constructed a new dataset from large-scale real-world projects (e.g., Chromium, Debian) and proposed a practical approach emphasizing representation learning to better distinguish vulnerable from non-vulnerable instances. VulDeePecker [23] pioneered deep learning-based binary vulnerability detection by leveraging LLVM intermediate representation to encode source code. Building upon this, Li et al. [24] proposed VulDeeLocator, an enhanced variant incorporating code attention mechanisms, which enables precise identification of vulnerability types even in scenarios with limited training samples.

2.3 Deep Learning with Graph-Based Code Representations

To preserve richer syntax and semantic information, many recent studies model source code as graphs. Representations such as ASTs, CFGs, and DDGs enable explicit capture of program structure. FUNDED [11] employs Graph Neural Networks (GNNs) to learn from program graphs integrating control, data, and call dependencies. VulDetector [12] introduces a Weighted Feature Graph (WFG) to emphasize vulnerability-sensitive nodes.

Recent studies further enhance expressiveness through multi-view representations. Wen et al. [25] proposes a code input representation to reduce node distances and applies enhanced graph learning, achieving strong results. Similarly, Qiu et al. [26] propose a multiple-graph framework that jointly models diverse program semantics, achieving state-of-the-art performance. Beyond accuracy, newer works focus on robustness and interpretability. Coca [27] improves and explains GNN-based detectors by training robust models and producing concise explanations. Luo et al. [28] introduce an efficient inter-procedural abstract graph and a heterogeneous attention GNN to exploit node/edge heterogeneity while maintaining scalability.

Despite their strength, graph-based methods face scalability challenges. Constructing and learning from full graphs, as in FUNDED and VulDetector, may cause graph size explosion and substantial overhead. Moreover, uniformly modeling all dependencies can introduce redundancy, potentially diluting vulnerability-relevant semantics and reducing robustness [29].

2.4 Large Language Models for Vulnerability Detection

Recently, Large Language Models (LLMs) have attracted increasing attention. By leveraging large-scale pre-training, LLMs demonstrate strong generalization in code understanding [30]. Shestov et al. [31] fine-tune a pre-trained code LLM (Wizard-Coder) for function-level detection, reporting improvements over CodeBERT-like baselines. Beyond general fine-tuning, recent studies explore domain-specific adaptation and systematic evaluation. RealVul [32] is an LLM-based framework tailored for PHP web vulnerabilities, using normalization strategies to isolate triggers. Meanwhile, Lin et al. [33] provide a comprehensive evaluation of LLM families for zero-shot detection, showing that model size, context window, and quantization substantially affect outcomes.

2.5 Motivation

Based on the analysis of existing methods, we identify that current vulnerability detection techniques face challenges in balancing structural semantics, efficiency, and generalization. While deep learning models have reduced the reliance on manual feature engineering, purely text-based approaches often fail to capture fine-grained control and data dependencies, which are critical for identifying complex vulnerabilities. Although graph-based methods preserve richer structural information, they frequently suffer from scalability issues and structural redundancy due to the complexity of full program graphs. Furthermore, recent large language models, while powerful, may act as black boxes and struggle with providing interpretable evidence for security auditing.

Motivated by these limitations, this study proposes VDDA, aiming to bridge the gap between structural precision and computational efficiency. We argue that an optimal representation should explicitly preserve vulnerability-relevant dependency paths while eliminating irrelevant code bloat. To this end, VDDA introduces a strategy that combines optimized CPGs with multi-directional slicing. This approach allows us to retain essential semantic and syntactic relationships without the overhead of processing entire program graphs. By focusing on localized, yet structurally rich, code blocks, VDDA establishes a more robust inductive bias for the downstream detection model, enabling it to learn vulnerability patterns more effectively and efficiently than existing state-of-the-art methods.

3 VDDA Software Vulnerability Detection Model

Fig. 1 summarizes the overall workflow of the proposed VDDA model. In this model, the source code is converted into CPGs by sequentially utilizing the Joern code analysis tool and the CPG optimization. Thereafter, by utilizing the program slicing and the code block formation methods, blocks of the source code are generated. Finally, by feeding code blocks to the tokenization and embedding modules in an orderly way, the vector representation of the source code is generated. The VDDA model is constructed by training the integration of the neural network and the attention mechanism.

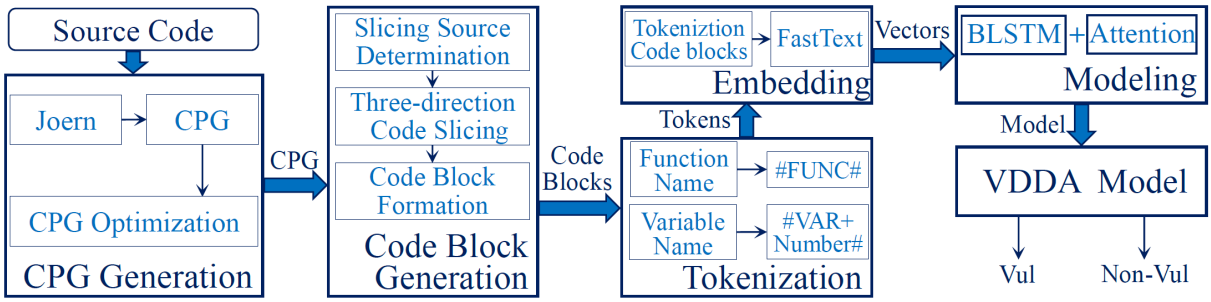


Fig. 1. Workflow of the VDDA vulnerability detection model

3.1 CPG Generation

Traditionally, to retain more syntax and semantic information, many deep models convert the source code into various types of graphs, such as AST, CDG, DDG, CFG, and PDG. Because it contains all the information in the above graphs, in this study, the CPG (generated by the Joern code analysis tool) is used to represent the target code. However, the abundant information of the preserved source code inevitably leads to the complex structure of the CPG. For example, the CPG shown in Fig. 2 is quite complicated, even for the simple sample code located in the upper-left corner of this figure. To improve the efficiency of deep learning vulnerability detection models, it is necessary to optimize the CPG before the resulting CPG is vectorized.

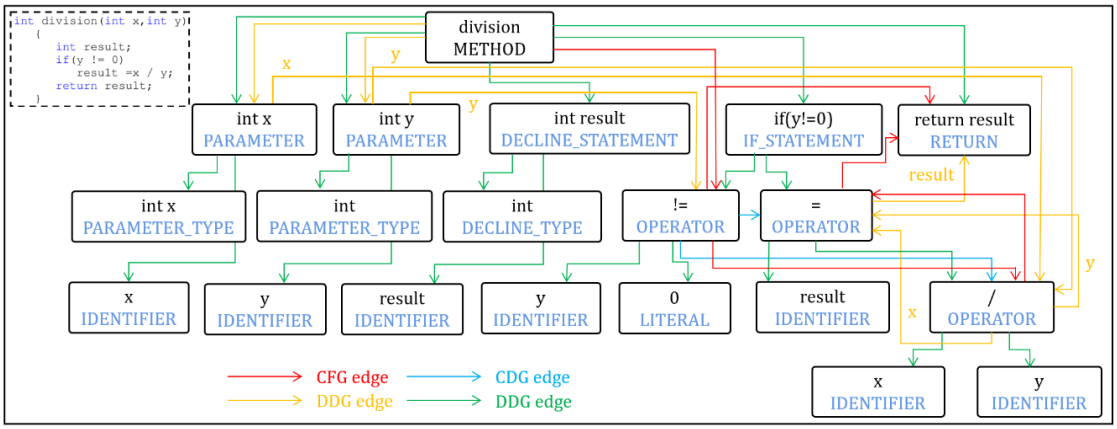


Fig. 2. CPG of a simple code sample

The CPG of an object function contains the edges of the CFG and CDG. Both edges specify control dependencies in the source code. Redundant information related to control dependencies degrades the performance of vulnerability detection. To improve the detection efficiency without reducing the vulnerability detection accuracy, a depth-first traversal-based CPG optimization algorithm is proposed to remove redundant CFG and CDG edges dynamically.

Assuming that $F = \{f_1, f_2, \ldots, f_n\}$ is the target program, and f_i is a sub-function of program F, the set of CPGs for the sub-functions of F generated using the Joern platform is $G = \{g_1, g_2, \ldots, g_n\}$, where g_i is the CPG of function f_i in the target program, and n_{ij} is the j^{th} node of g_i. The main steps of the CPG optimization algorithm are presented in *Algorithm* 1 (Fig. 3).

To preserve the control dependency information, *Algorithm* 1 selects only one of the CFG edges or CDG edges when both edges appear between two nodes in the CPG. *Algorithm* 1 performs depth-first traversal on the edges of CPG g_i in G. During the process of traversal on the edges of g_i, if a CFG edge connecting nodes n_{ij} and n_{ik} is encountered first, this edge is recorded in the CFG. Thereafter, the CDG edge between nodes n_{ij} and n_{ik} is removed during the remaining traversal of the CPG. Similarly, if a CDG edge connecting nodes n_{ij} and n_{ik} is encountered first, this edge is recorded in the CDG. Thereafter, the CFG edge between nodes n_{ij} and n_{ik} is removed during the remaining traversal of the CPG.

Algorithm 1 CPG optimization

Input: The set of CPGs obtained after the object program F is processed with the Joern tool, $G = \{g_1, g_2, \ldots, g_n\}$.

Output: G_1 //Set of compact CPGs after removing CFG edges or CDG edges from CPGs,

```
G1 ←∅;
for Each gi in G do
    for nodes nij, nik in gi do
        if edge<nij,nik> in CDG then
            if <nij,nik> is in CFG then
                Remove <nij,nik> from CFG;
            end if
        end if
        if edge<nij,nik> in CFG then
            if <nij,nik> is in CDG then
                Remove <nij,nik> from CDG;
            end if
        end if
    end for
    gi →G1;
end for
```

Fig. 3. CPG optimization algorithm

3.2 Code Block Generation

Since not all program statements are associated with vulnerabilities, scanning the entire source code of the target program is often unnecessary. Under typical circumstances, vulnerability-related code constitutes only a small portion of the overall codebase. For instance, the only difference between normal code and its vulnerable counterpart with the same functionality may be as minor as a single variable declaration. Therefore, it is crucial to simplify the target code before inputting it into underlying deep learning models. To enable finer and more flexible vulnerability detection, code slicing is performed prior to generating code blocks.

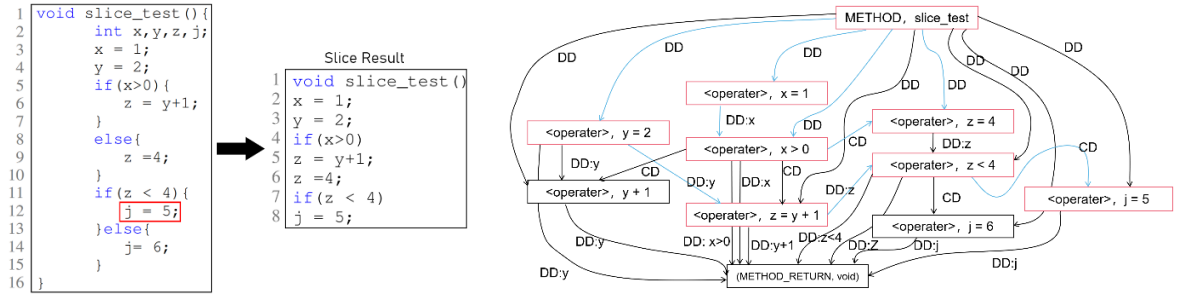
Slicing Source Determination. Different selections of slicing sources typically yield varying slicing results. Given that most software vulnerabilities are closely related to API function calls and arithmetic expressions (AEs), code lines involving API calls and AEs are selected as primary slicing sources. Specifically, any token nodes in the CPG corresponding to AST elements that represent APIs or AEs are considered valid slicing sources. For example, the *strncpy()* function on Line 21 in the sample code of Fig. 4 is an API function that may result in a buffer overflow vulnerability. Consequently, the variable data on Line 23, denoted as <23, data> in VDDA, can be used as a slicing source. Additionally, any code difference observed between the vulnerable and the patched versions can also be used as a slicing source.

Three-direction Code Slicing. In the CPG, both data and control dependency information are encoded in the edges connecting different nodes. The type of slicing performed depends on the slicing source. For function call and AE sources, where the vulnerable code is typically executed before the function call or arithmetic operation, backward slicing is applied. For sources related to patched code or pointer declarations, forward slicing is conducted. In the case of composite or other ambiguous slicing sources, both forward and backward slicing are executed simultaneously to retain as much syntactic and semantic context as possible.

The slicing procedure starts from the selected slicing sources, removing all irrelevant statements while preserving those that exhibit control or data dependencies with the slicing sources. For instance, in Line 12 of Fig. 5(a), the statement "*j*=5" is selected as the slicing source. Backward slicing is performed on the *slice_test()* function, and the resulting code slice and its optimized CPG are shown on the right side of Fig. 5(a) and Fig. 5(b), respectively.

```
1  void printLine(const char * line) {
2     if(line!=NULL)
3     {
4        printf("%s\n",line);
5     }
6  }
7  void func_name()
8  {
9     char *data;
10    char *dataBadBuffer= (char*)
          ALLOCA(50*sizeof(char));
11    char *dataGoodBuffer=(char*)
          ALLOCA(100*sizeof(char));
12    if(GLOBAL_CONST_FIVE==5)
13    {
14       data = dataBadBuffer;
15       data[0] = '\0';
16    }
17    {
18       char source[100];
19       memset(source, 'C', 100-1);
20       source[100-1] = '\0';
21       strncpy(data, source, 100-1);
22       data[100-1] = '\0';
23       printLine(data);
24    }
25 }
```

Fig. 4. Sample code for slicing source determination



(a) Slicing result of sample code (b) CPG of slicing result

Fig. 5. CPG and slicing result of sample code

Code Block Generation. In contrast to existing vulnerability detection approaches that directly tokenize raw slices, VDDA further processes the generated slices into informative code blocks. This additional processing is necessary because different slicing directions (e.g., backward vs. forward) place vulnerability-relevant statements at different relative positions, and such positional semantics can affect downstream sequence-based models.

For each slicing source s, VDDA first collects all slices generated from s under different slicing directions. To ensure reproducibility, VDDA employs an *order-preserving criterion* to remove duplicate slices. Specifically, each slice is represented as an ordered statement sequence $S(s,d)=\langle n_1,n_2,\ldots,n_k\rangle$, where n_i denotes a statement node (or a source-code statement) preserved by slicing, and the order follows the original source-code order (e.g., line numbers). Two slices are considered duplicates if and only if their ordered sequences are identical, i.e., $S(s,d_i)=S(s,d_j)$. In implementation, this can be checked by hashing the concatenation of statement-node IDs (or normalized statement strings) and keeping only one instance per hash.

After deduplication, VDDA assembles direction-aware code blocks. If a single direction is applied, the corresponding slice sequence is directly used as the code block. If both backward and forward slices are used, VDDA merges them at the slicing source while removing cross-direction overlap to avoid repeated statements. Specifically, let $B(s)$ and $F(s)$ denote the ordered backward and forward slices of source s, respectively. Since the slicing-source statement appears in both directions, VDDA constructs the merged code block according to the following rule:

$$B(s)\oplus F(s)-\{s\}, \tag{1}$$

where $\oplus$ denotes concatenation and $\{s\}$ denotes the slicing-source statement kept only once. This guarantees that each statement appears at most once in the final code block while preserving the direction-induced positional semantics. Fig. 6 illustrates the above procedure using *slice_test()* with slicing source A ($S7$: j=5):

(1) *Backward Slice*: Retains statements that affect whether $S7$ is executed or how its context is formed, including control dependencies (e.g., $S7$ depends on $S6$: z<4, which is constrained by the else path of $S3$: x>0) and data dependencies (e.g., $S6$ uses z defined at $S5$: z=4). Thus, $Backward(A)=[S0,S1,S3,S5,S6,S7]$.

(2) *Forward Slice*: Degenerates to $Forward(A)=[S7]$ because j is not used afterward.

(3) *Merging*: The overlap $Backward(A)\cap Forward(A)=\{S7\}$ is removed, yielding $BiDir(A)=Backward(A)\oplus(Forward(A)-\{S7\})=[S0,S1,S3,S5,S6,S7]$. In this case, $BiDir(A)$ is identical to $Backward(A)$ and is treated as a duplicate, leaving only $Backward(A)$ and $Forward(A)$ for subsequent code block construction. The same process applies to slicing source B ($S8$: j=6), as also shown in Fig. 6.

Code Block Labeling. After code blocks are generated, VDDA assigns labels to enable supervised learning. Each code block inherits the ground-truth label of its originating code unit (e.g., file/function) in the dataset: vulnerable blocks are labeled as 1 and non-vulnerable blocks as 0. In practice, many public benchmarks provide file-level labels through naming conventions. For example, in the NVD dataset, a source file tagged with the keyword “VULN” indicates a vulnerable version, whereas a file tagged with “PATCHED” indicates a non-

vulnerable (patched) version. Fig. 7 shows a concrete example of this convention. VDDA maps this file-level label to all code blocks generated from the same file, thereby producing the final training samples in the form of (*code block, label*).

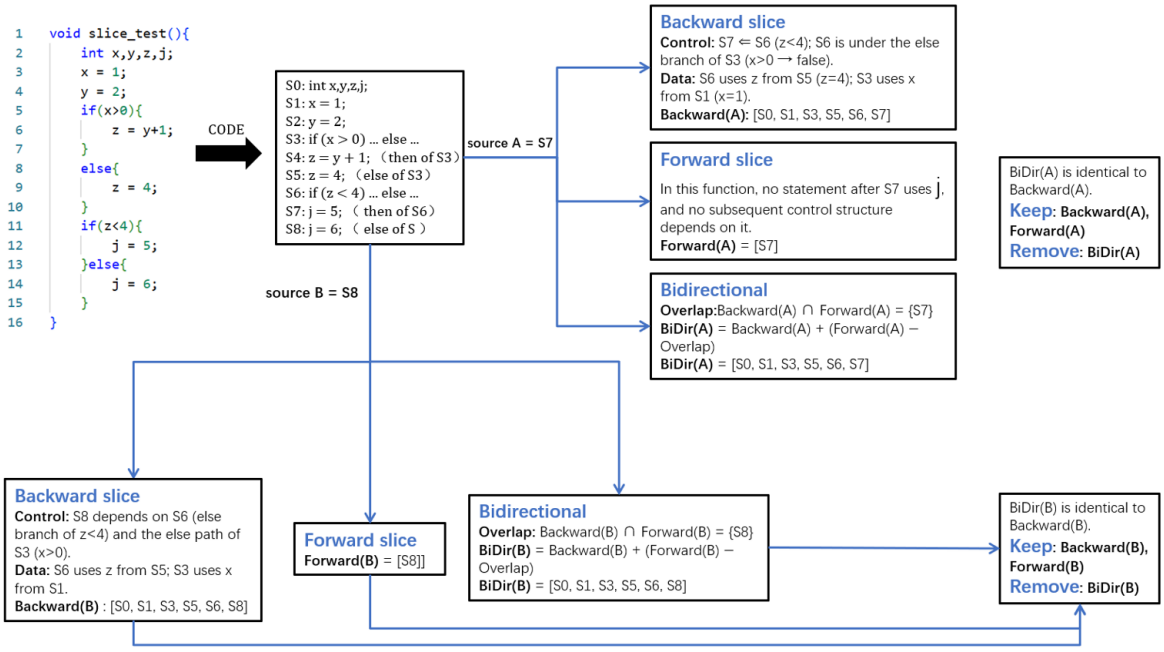


Fig. 6. An example of three-direction slicing and deduplication

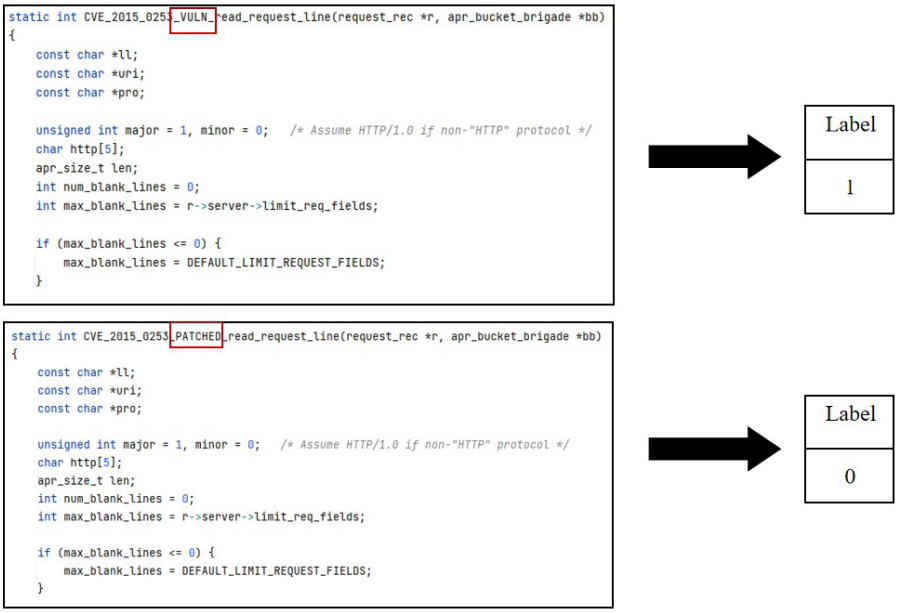


Fig. 7. Code labeling in the NVD dataset

3.3 Code Tokenization

Under normal circumstances, the name of a function and the variable names in the body of this function do not significantly affect the vulnerability detection performance. Different functions may contain the same type of vulnerability. To reduce the interference caused by the uncertainty of names in the training process, function names and variable names in the function body are symbolized before the code blocks are tokenized. Specifically, all the function names in a single slice are normalized as "#FUNC#", and all the variable names in the body of the corresponding function are normalized as "#VAR+number#".

After symbolization, token splitting is performed. In this stage, lexical analysis is used in the process of tokenization. As shown in Fig. 8, lexical analysis scans the target program and splits each statement into smaller tokens.

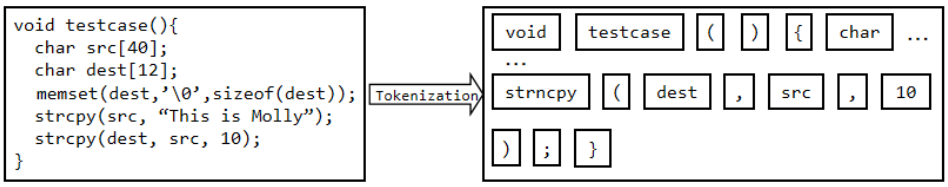


Fig. 8. Example of tokenization

3.4 Code Vectorization

It is necessary to convert the generated code blocks into vectors to enable the use of deep learning frameworks. Owing to its stable performance and strong anti-interference (such as the disturbances of typos), FastText [34] was used to vectorize code blocks in this study. Specifically, the tokens obtained are first collected into a corpus. Tokens are then taken as the input to FastText to generate the vector for each token. Finally, the vectors of the tokens in a single code block are collected to complete the vectorization. However, because of the different code lengths in the code blocks, the number of split tokens is different. The lengths of the generated vectors must be normalized.

It is well known that the closer a piece of code is to the slicing source, the more effective is the information it contains. In this study, two positions are defined to specify the importance of different part of code blocks:

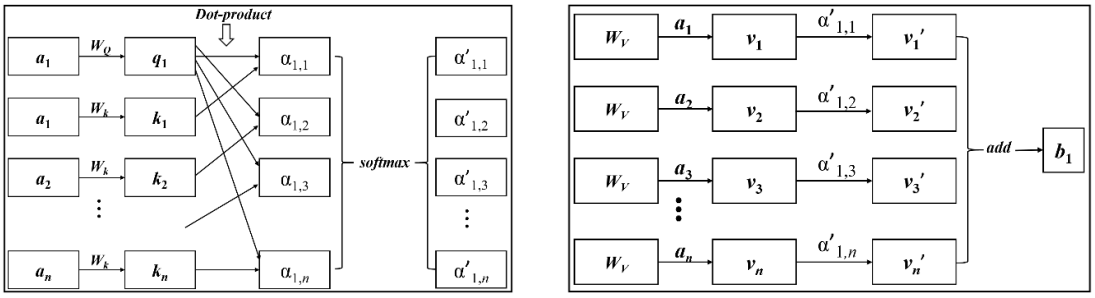
(1) *Expensive-end*, denoted by *ep*. It specifies code positions that are close to the slicing source.

(2) *Cheap-end*, denoted by *cp*. It specifies code positions that are far from the slicing source.

As discussed in the previous section, three directions, backward, forward, backward and forward combined, are incorporated in the process of code slicing. For forward slicing, *ep* is at the beginning of the vector and *cp* is at the end of the vector. The *ep* and *cp* of backward slicing are opposite to those of forward slicing. Let us consider forward slicing as an example. When the length of a vector is less than the specified value (this value is generally assigned according to the size of the code blocks), the *cp* end is padded with 0. In contrast, tokens at the ep end are pruned to satisfy the specified length. For bidirectional direction slicing, the *ep* end is the position of the slicing source, and both the starting point and the end point are at the *cp* end where 0 can be used as padding or tokens can be pruned.

3.5 Formation of the VDDA Model

Because most vulnerabilities are not caused by just one code line but are closely related to the context, BLSTM is used to detect software vulnerabilities. In a code block, the importance of different slices to vulnerability detection is not the same. Meanwhile, the importance of code statements and tokens in a slice also differs. If all these components are treated equally, the vulnerability detection performance degrades. In this study, the attention mechanism is incorporated into the deep learning process. In the previous stage, code blocks and vectorized tokens in these blocks were obtained. The attention mechanism considers the relationships among different code blocks or tokens in a single code block.



(a) Formation of attention score (b) Formation of final result vector

Fig. 9. Method for obtaining the attention score

At the token level, it is supposed that vectors, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_n$, are generated by tokens, $\boldsymbol{token}_1$, $\boldsymbol{token}_2$, …, $\boldsymbol{token}_n$, respectively. As shown in Fig. 9(a), by multiplying vectors, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_n$, with the query matrix, $\boldsymbol{W}_Q$, and the key matrix, $\boldsymbol{W}_K$, in an orderly way, the corresponding query vector $\boldsymbol{q}_1$ and the key vectors, $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, …, $\boldsymbol{k}_n$, are first obtained. The attention score, denoted by α, between any of the two tokens is then obtained by performing the *dot product* operation on the two types of vectors. The attention score specifies the correlation between two tokens; the higher the value of α, the stronger the correlation between them. Finally, as shown in Equation (1), the value of α is normalized by *softmax*.

$$\alpha'_{x,y} = \frac{\exp(\alpha_{x,y})}{\sum_{j=1}^{n}\exp(\alpha_{x,j})}, \quad x, y \in [0, 1]. \tag{2}$$

In Equation (2), $\alpha'_{x,y}$ species the relevance (i.e., the attention score) between the x^{th} vector and the y^{th} vector in $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n\}$.

In Fig. 9(b), as in Equation (3), the value vectors, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, …, $\boldsymbol{v}_n$, of the original token vectors are obtained by multiplying vectors, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_n$, with the value matrix $\boldsymbol{W}_V$ (randomly generated) in an orderly way.

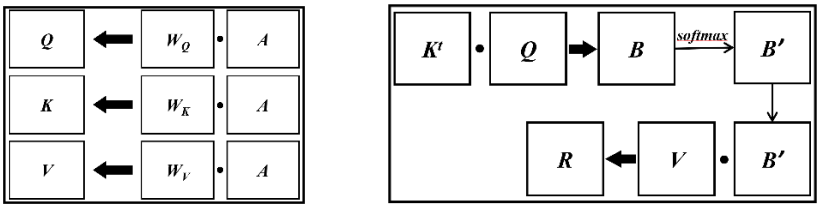
$$\boldsymbol{v}_i = \boldsymbol{W}_V \cdot \boldsymbol{a}_i, \quad i \in [0, 1]. \tag{3}$$

The final result vector $\boldsymbol{b}_x$ is obtained by Equation (4).

$$\boldsymbol{b}_x = \sum_{i=1}^{n} \alpha'_{x,i} \cdot \boldsymbol{v}_i, \quad i \in [0,1]. \tag{4}$$

The method used to obtain the attention score at the code block level is the same as that used at the token level. In general, as shown in Fig. 10(a), by separately multiplying the embedding matrix $\boldsymbol{A}$ (composed of token vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_n$) with the query matrix $\boldsymbol{W}_Q$, key matrix $\boldsymbol{W}_K$, and randomly generated value matrix $\boldsymbol{W}_V$, three vector spaces, that is, $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, are first obtained. Then, as shown in Fig. 10(b), by performing a dot product operation between $\boldsymbol{K}_t$ ($\boldsymbol{K}_t$ is the transpose of matrix $\boldsymbol{K}$) and $\boldsymbol{Q}$ ($\boldsymbol{K}_t \cdot \boldsymbol{Q}$), the attention score matrix $\boldsymbol{B}$ is obtained. The final result vector $\boldsymbol{R}$ is obtained by normalizing the attention score matrix $\boldsymbol{B}$ first and multiplying with $\boldsymbol{V}$ second.

The aforementioned processes are executed twice. In the first execution, the embedding matrix $\boldsymbol{A}$ is composed of token vectors. In the second execution, the embedding matrix $\boldsymbol{A}$ is composed of code block vectors. Using these two executions, a set composed of optimized result vectors is obtained.



(a) Vector space (b) Result matrix

Fig. 10. Vector spaces and result matrix generation

4 Experimental Results

In this section, experiments to determine the code block size, embedding method, symbolic algorithm, dimension of token vectors, and underlying deep learning model are first performed to obtain the optimal parameters for the proposed VDDA model. Subsequently, the effectiveness of the CPG optimization algorithm is tested. Third, an ablation experiment is performed to test the contributions of different parts of the VDDA to vulnerability detection. Finally, the overall performance of the VDDA model is compared with that of eight existing vulnerability detection methods.

The computer that ran these experiments comprised an Intel(R) Xeon(R) Gold 6330 CPU and an NVIDIA RTX 3090 GPU. The dataset, denoted as *N&S*, used in the experiments was collected from the NVD [35] and SARD [36]. Among the 12,224 files in *N&S*, 2022 files were selected from the open-source project NVD, and 10,202 files were selected from the buffer overflow source code vulnerability dataset SARD. Eighty percent of the files from the built dataset were randomly selected as the training set. The remaining 20% of the files were used as the testing set. Additionally, 30% of the files from the training set were selected as the validation set.

To verify the generalization capability of the proposed VDDA model, another dataset, denoted as *D&S*, was tested in an experiment that compared the overall performance of the VDDA with that of existing vulnerability detection models. The *D&S* dataset was built from two public vulnerability datasets, that is, D2A [37] and S-Babi [38]. The program samples in S-Babi were from artificially synthesized code. The program samples in the D2A dataset were from real application code. The total 7014 samples in the D&S testing dataset were composed of 3012 samples from D2A and 4000 samples from S-Babi.

Four metrics, *Accuracy*, *Precision*, *Recall*, and F_1-*Score*, were used to evaluate the vulnerability detection performance of the proposed model.

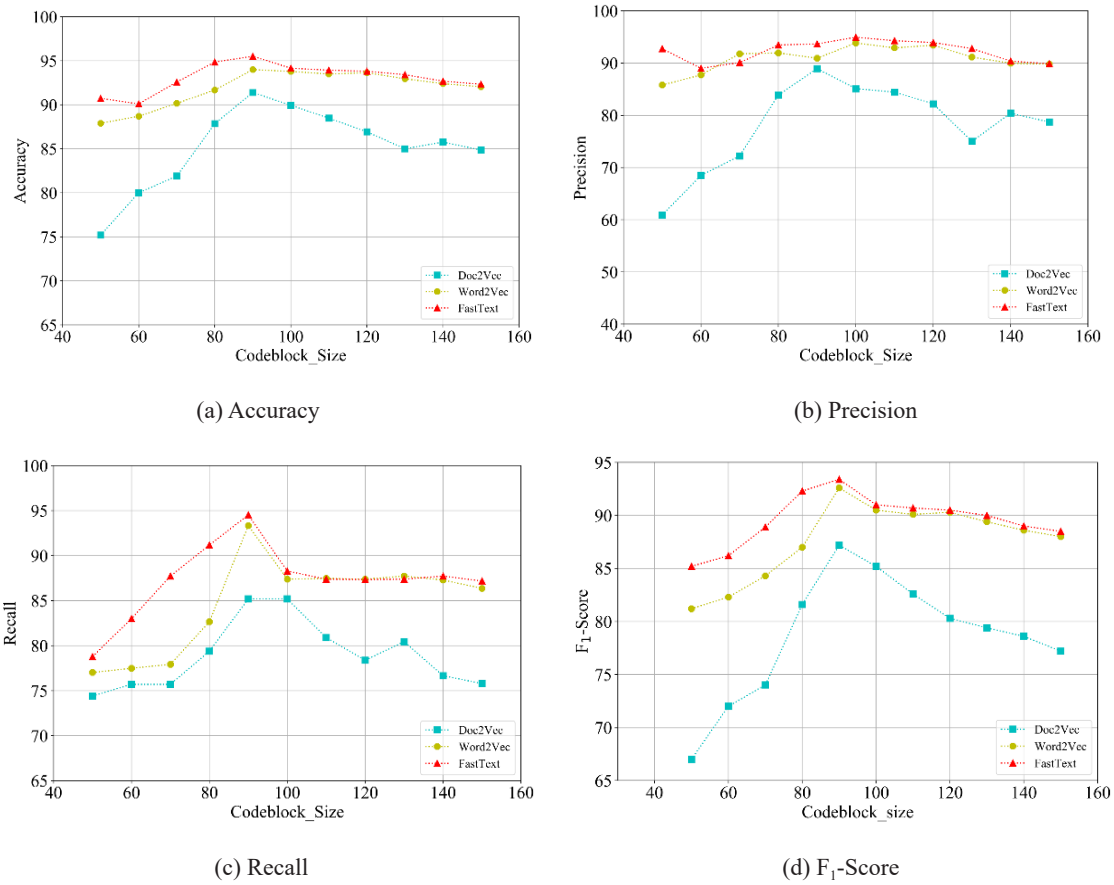
4.1 Code Block Size and Embedding Method Determination

Different code block sizes and word embedding methods have varying impacts on vulnerability detection performance, primarily because they jointly determine: (1) the extent to which contextual information around vulnerability triggers can be preserved; and (2) whether code tokens are mapped into the vector space in a sufficiently stable and reliable manner. In this experiment, we vary the code block size from 50 to 150, and comparatively evaluate three commonly used embedding methods (Doc2Vec, Word2Vec, and FastText), with results presented in Fig. 11.

Overall, FastText demonstrates the most stable performance across different block sizes and achieves the best overall results. This is largely attributable to the fact that real-world source code contains a large number of out-of-vocabulary (OOV) identifiers (e.g., project-specific variable or function names) and low-frequency tokens, along with common minor perturbations such as naming style variations and spelling errors. Unlike Word2Vec and Doc2Vec, FastText leverages subword (character *n*-gram) information for representation learning which enables the model to not only generate more robust vector representations for low-frequency or unseen tokens, but also provide greater fault tolerance against minor token perturbations. Consequently, FastText maintains more consistent representation capability when code token distributions are sparse or contain noise.

Regarding code block size, we observe a clear trade-off between semantic completeness and noise/computational overhead. When the block size is too small (e.g., 50-70), code blocks may fail to include sufficient contextual statements to cover the dependency information required for vulnerability triggering, leading to performance degradation. As the block size increases, model performance gradually improves and reaches optimal levels around 90 across most metrics and embedding methods, indicating that this size provides adequate contextual coverage while avoiding excessive redundant information. Although *Precision* for FastText and Word2Vec peaks at a block size of 100, larger blocks tend to introduce more weakly relevant or irrelevant statements, which increases noise and overfitting risks and significantly raises time and memory overhead during preprocessing and training

Considering overall performance as well as the balance between *Accuracy* and efficiency, this study ultimately sets the code block size for VDDA at 90 and selects FastText as the embedding method.



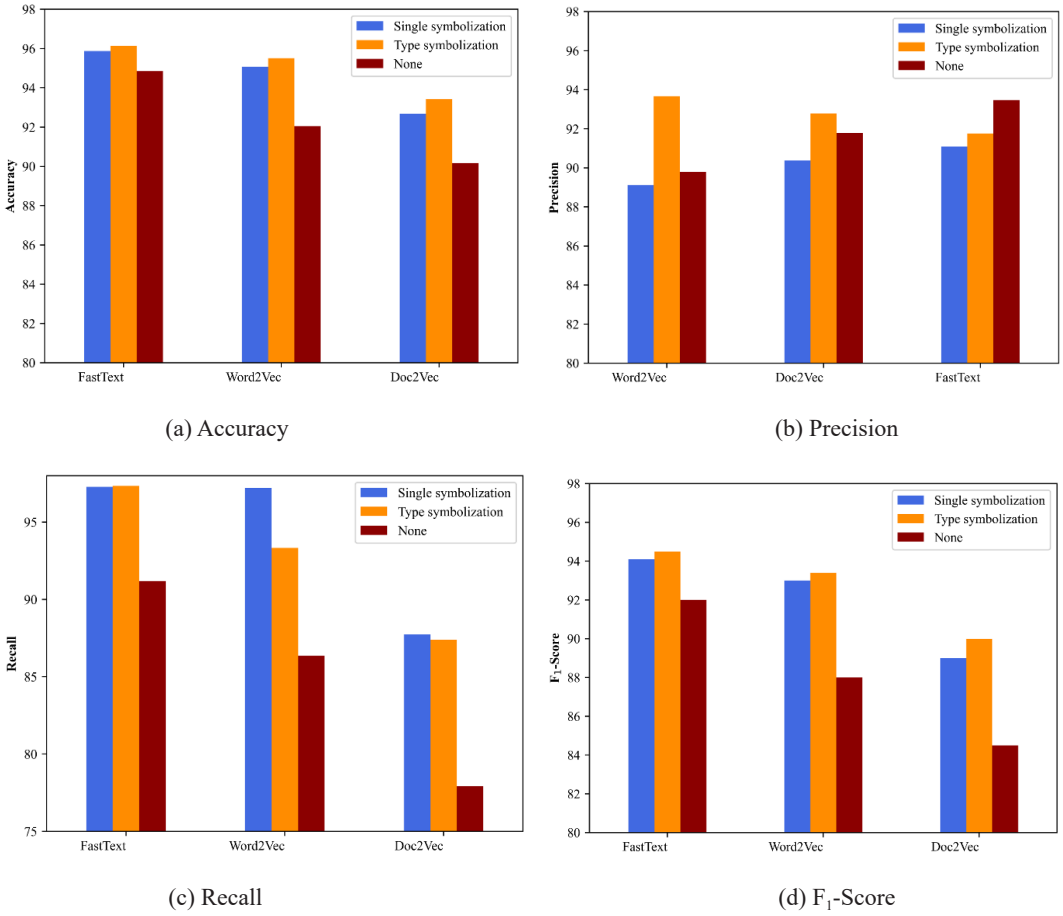
(a) Accuracy (b) Precision

(c) Recall (d) F_1-Score

Fig. 11. Experiment on determining the size of code blocks

4.2 Determination of Symbolization Method

Prior to tokenizing code blocks, it is necessary to perform symbolization on function names and variable names within function bodies. This process aims to mitigate the sparsity caused by project-specific identifiers while preserving vulnerability-relevant semantic information as much as possible. In real-world source code, user-defined function names and variable names exhibit high diversity and often appear as low-frequency or even OOV tokens. Directly learning these identifiers may cause models to overfit to superficial naming patterns rather than the underlying program behavior semantics.



(a) Accuracy (b) Precision

(c) Recall (d) F_1-Score

Fig. 12. Experiment on determining the symbolization method

To address this challenge, we propose the *Type Symbolization* strategy, which treats function names and variable names differently to preserve their distinct semantic roles. Specifically, all function names within the same slice are normalized to "#FUNC#", while variable names in the corresponding function body are normalized to "#VAR+number#". We also record variable data types to retain type-level information cues, as type information often holds significant value for vulnerability detection.

To evaluate the effectiveness of *Type Symbolization*, we compare it with the commonly used *Single Symbolization* strategy and set no symbolization (*None*) as the baseline. In *Single Symbolization*, both function names and variable names are uniformly normalized. This experiment fixes the code block size at 90 and uses LSTM as the underlying neural network. We test different symbolization strategies under three embedding methods (Doc2Vec, Word2Vec, FastText) to analyze whether the benefits remain consistent across different vectorization approaches.

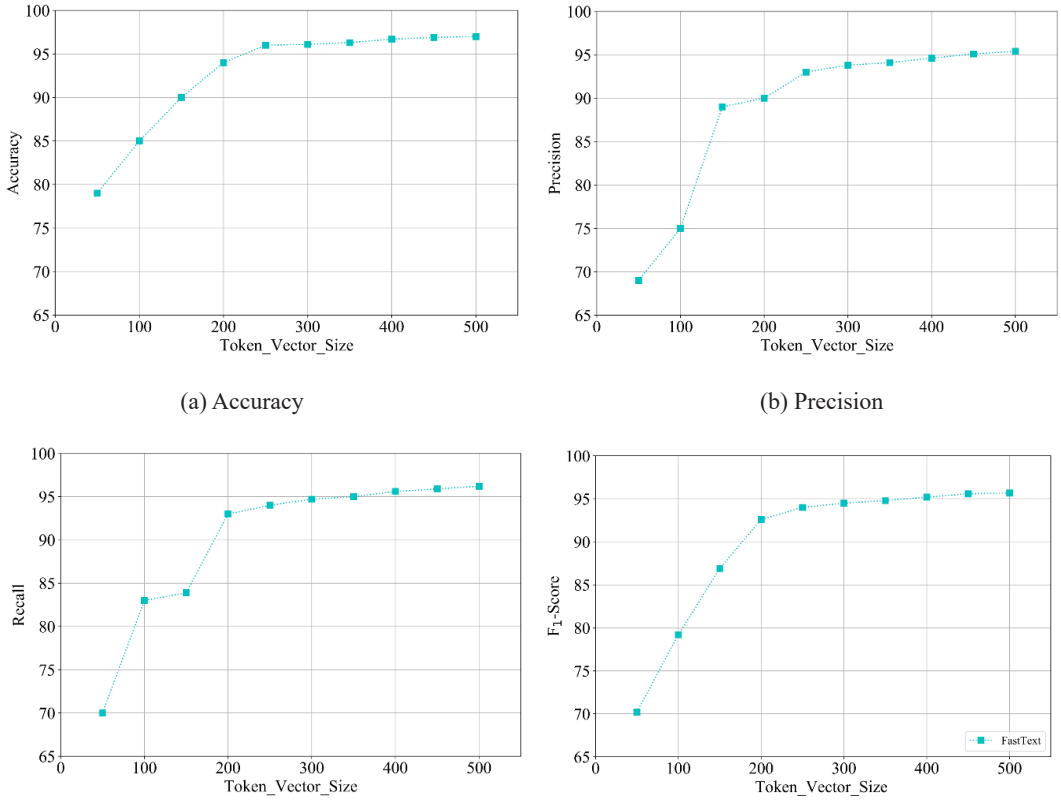
As shown in Fig. 12, *Type Symbolization* generally outperforms both *Single Symbolization* and *None*, indicating that distinguishing between identifier categories helps models learn more stable and transferable vulnerability patterns. This improvement stems from the different semantic roles played by function names and variable names: function names primarily correspond to operation semantics or API usage, while variable names are more closely related to data flow and state. Uniform processing of both types tends to blur their semantic role differences, thereby weakening the model's ability to model vulnerability-relevant interaction relationships (e.g., how a particular operation acts on specific data objects). Additionally, preserving type information further reduces semantic ambiguity and provides additional constraints for model learning (e.g., distinguishing between pointer variables and integer variables), which is particularly beneficial for identifying memory-related and type-related vulnerabilities.

We also observe that the combination of *Type Symbolization* and FastText achieves the best performance. This can be attributed to FastText's subword modeling capability, which provides more robust representations for low-frequency tokens and demonstrates greater stability against minor lexical perturbations, thus complementing

the symbolized corpus. Overall, the experimental results demonstrate that *Type Symbolization* generates higher-quality corpora for subsequent embedding and learning, leading us to adopt this strategy in VDDA.

4.3 Token Vector Dimension Determination

The dimensionality of token vectors is a critical hyperparameter, as it directly influences both the model's representational capacity and the computational overhead during training and inference. An excessively small dimension may fail to encode sufficient discriminative semantic information. Conversely, an overly large dimension significantly increases time and memory consumption and may introduce redundant representations. In this experiment, we fix the vectorization pipeline established in the previous sections (using FastText with a code block size of 90 and *Type Symbolization*) and vary the token vector dimension from 50 to 500 to identify a reasonable compromise between performance and resource cost.



(a) Accuracy (b) Precision

(c) Recall (d) F_1-Score

Fig. 13. Experiment on determining the dimension of token vectors

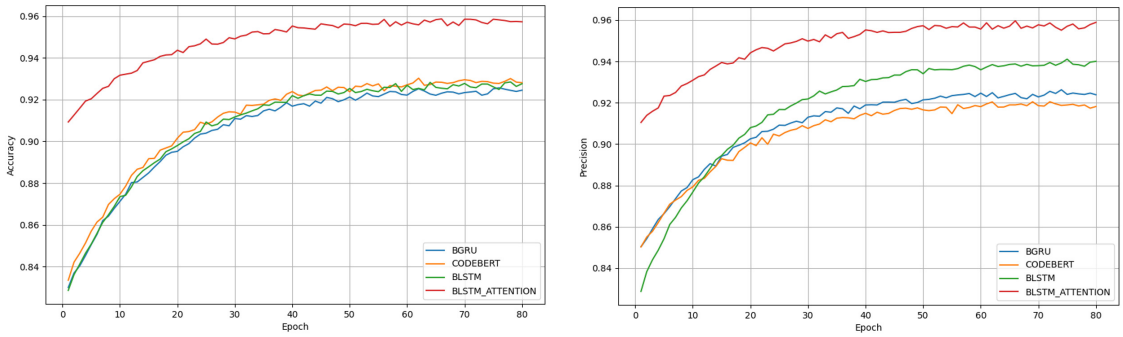
As illustrated in Fig. 13, the metrics, including *Accuracy*, *Precision*, *Recall*, and *F_1-Score*, exhibit an upward trend as the vector dimension increases. This indicates that higher-dimensional embeddings can capture richer lexical and contextual distinctions from the symbolized code corpus. However, beyond approximately 300 dimensions, the rate of improvement for all metrics noticeably slows down, and the curves enter a state of "saturation". This suggests that, given a fixed dataset size, the additional representational capacity offered by higher dimensions cannot be fully utilized by the model, leading to diminishing returns.

Simultaneously, higher dimensions significantly expand the memory footprint of the embedding matrix and intermediate activations, while also increasing training and inference latency, rendering them inefficient from an engineering deployment perspective. Therefore, after carefully weighing the trade-off between detection efficacy and computational efficiency, we set the token vector dimension for VDDA to 300. This setting maintains strong detection performance while avoiding unnecessary resource overhead.

4.4 Deep Learning Model Determination

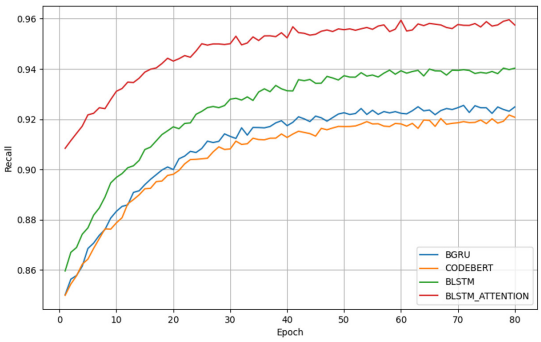
Software vulnerabilities are rarely caused by isolated statements. Instead, they are typically triggered by contextual dependencies spanning multiple statements, such as the interaction between conditional checks, variable assignments, and security-sensitive operations. Therefore, the underlying deep learning model must be capable of capturing long-range contextual cues within the structured code blocks generated by the optimized CPG and three-direction slicing.

In this experiment, we evaluate two bidirectional recurrent neural network architectures, namely the BLSTM and the Bidirectional GRU (BGRU), and further investigate the impact of integrating an attention mechanism into the BLSTM (BLSTM-Attention). Additionally, we include the representative pre-trained language model, CodeBERT, as a comparative baseline. To ensure a fair comparison, all models are trained under identical data splits and experimental settings, including the same input representations, code block construction strategies, and CPG optimization configurations.

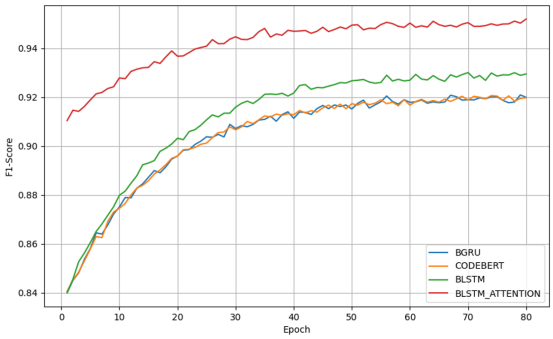


(a) Accuracy

(b) Precision



(c) Recall



(d) F_1-Score

Fig. 14. Experiment on determining the underlying neural networks

As shown in Fig. 14, BLSTM outperforms both BGRU and CodeBERT across all four metrics. This superiority may be attributed to BLSTM's gating and memory mechanisms, which, combined with bidirectional modeling, are particularly effective at capturing the long-range contextual dependencies inherent in code. Although BGRU is computationally lighter, it may exhibit slightly weaker expressive power regarding fine-grained long-range interactions under the same parameter budget. In contrast, CodeBERT primarily learns representations from linearized token sequences. Given that our input consists of "dependency-guided structured code blocks" derived from the optimized CPG, a structural semantic misalignment occurs, potentially limiting its effectiveness in this specific setting.

More critically, the integration of the attention mechanism yields a significant performance boost. This observation suggests that different statements within a code block contribute unequally to vulnerability detection. Vulnerability-triggering semantics are often concentrated around a few key statements (e.g., validation checks, taint sources, and dangerous operations), while the majority of contextual statements may be auxiliary or

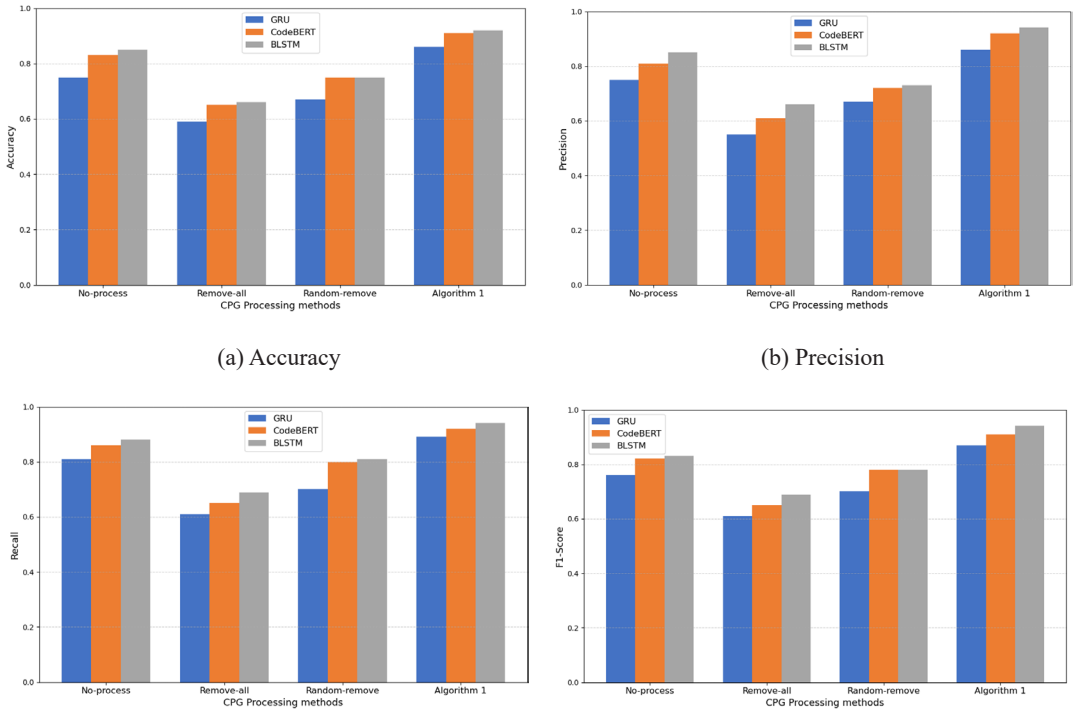
weakly related. The attention mechanism addresses this by assigning higher weights to critical statements and suppressing irrelevant context, thereby enhancing feature selectivity and robustness. Based on these results, this paper adopts BLSTM-Attention as the underlying vulnerability detection model for VDDA.

4.5 Performance of the CPG Optimization Algorithm

The CPGs inherently contain redundancy between CFG edges and CDG edges. To address this, we propose *Algorithm* 1 (presented in Fig. 3) to optimize the generated CPG. This algorithm selectively removes redundant control-related edges to reduce the structural complexity of the graph.

To validate its effectiveness, we compare *Algorithm* 1 against three baseline strategies: (1) *No-process*: retaining the original CPG without modification; (2) *Remove-all*: deleting both CFG and CDG edges when multiple edges exist between the same pair of nodes; and (3) *Random-remove*: randomly retaining one edge while discarding the other when overlap occurs.

Under a consistent experimental pipeline, we apply these four CPG processing strategies to three representative models (GRU, BLSTM, and CodeBERT) and evaluate their vulnerability detection performance. As illustrated in Fig. 15, *Algorithm* 1 achieves superior results across all four metrics. This comparison demonstrates that "heavy-handed" edge removal strategies are inadequate: *Remove-all* may inadvertently discard discriminative control semantics, thereby weakening the model's ability to reason about vulnerability triggers, while *Random-remove* introduces instability across samples due to the uncertainty in which semantic information is preserved.



(a) Accuracy (b) Precision

(c) Recall (d) F_1-Score

Fig. 15. Comparisons of different CPG processing methods

In contrast, *Algorithm* 1 eliminates redundancy in a more structured manner. By suppressing unnecessary graph inflation while better preserving dependency cues relevant to vulnerability detection, it effectively improves the signal-to-noise ratio of the graph representation. Overall, the results indicate that appropriate pruning of redundant CFG/CDG edges not only preserves detection accuracy but enhances it. Furthermore, the reduction in graph complexity decreases the overhead associated with graph construction and traversal, lowers the effective input complexity for downstream learning, and consequently contributes to more efficient, stable, and reliable detection performance.

4.6 Ablation Experiment for VDDA

Based on the preceding analyses, the final configuration of VDDA is established as follows:

(1) *CPG Optimization*: *Algorithm* 1 is applied to optimize the CPG generated by the Joern code analysis platform.
(2) *Code Slicing*: Optimized CPGs are organized into code blocks of size 90 using the three-direction slicing method.
(3) *Symbolization*: The *Type Symbolization* method is employed to process function and variable names within the function bodies.
(4) *Embedding*: The FastText method converts tokenized code blocks into 300-dimensional vectors.
(5) *Model Architecture*: BLSTM integrated with an attention mechanism serves as the underlying deep learning model.

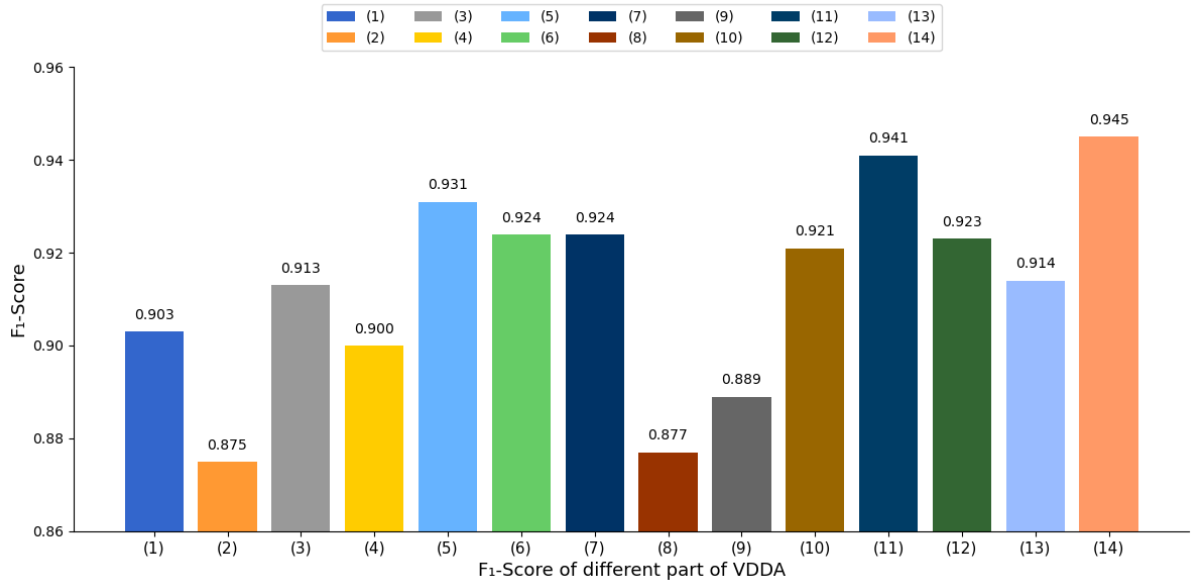


Fig. 16. Ablation experiment for VDDA

Fig. 16 shows the results of the ablation experiment for VDDA on the *N&S* dataset. As the harmonic average of *Precision* and *Recall*, the F_1-*Score* is better than the other evaluation indices. In this experiment, the vulnerability detection performance is tested using the F_1-*Score* to save space.

As shown in Fig. 16, the 14th bar provides the overall VDDA performance (denoted as (14) in this figure, the same as below). The 1st and 2nd bars represent the performance of VDDA when *Algorithm* 1 is replaced with *No-process* (1) and *Random-remove* (2), respectively. The experimental results demonstrate that *Algorithm* 1 contributes more to the overall performance of VDDA than do *No-process* and *Random-remove*. Compared with the *No-process* and *Random-remove* CPG processing methods, *Algorithm* 1 improves the performance of VDDA by 2.5% and 4.2%, respectively.

The 3rd and 4th bars represent the VDDA performance when the size of the code block from 90 to 80 (3) and 100 (4) is replaced, respectively. The experimental results demonstrate that a large code block does not always achieve better performance in vulnerability detection. A large block size not only requires considerable time and memory resource consumption by the underlying platform, but also induces overfitting in the deep learning model.

The 5th - 9th bars represent VDDA performance when *Type symbolization* on FastText is replaced by *Single symbolization* on FastText (5), *Type symbolization* on Word2Vec (6), *Single symbolization* on Word2Vec (7), *Type symbolization* on Doc2Vec (8), and *Single symbolization* on Doc2Vec (9). The experimental results demonstrate that the combination of *Type symbolization* and FastText achieves the best vulnerability detection performance.

The 10th and 11th bars represent the VDDA performance when the size of the token vector from 300 to 200 (10) and 400 (11) is replaced, respectively. In fact, the longer the token vector, the better the vulnerability detection performance. However, after the size of the dataset is fixed, the benefits brought about by only increasing the dimension cannot offset the negative effects caused by the increase in time and memory resources.

The 12th and 13th bars in the chart represent the performance of VDDA when the attention-enhanced BLSTM is replaced by a vanilla BLSTM (12) and CodeBERT (13), respectively. The performance degradation observed in both ablation studies indicates that: (1) the attention mechanism assists the BLSTM in focusing on vulnerability-relevant statements within the structured code blocks; and (2) in the context of our graph-guided detection framework, sequence-based pre-trained encoders are not necessarily optimal.

Although CodeBERT demonstrates strong capability in representation learning over linearized code through general pre-training objectives (e.g., Masked Language Modeling) and token-level co-occurrence patterns, it struggles to explicitly capture the cross-statement control/data dependency paths (such as sanitization checks, boundary validations, or taint propagation chains) that determine whether a vulnerability is actually triggered. In contrast, VDDA leverages CPG optimization and three-direction slicing to retain and localize vulnerability-relevant dependencies while pruning redundant edges. This process establishes an inductive bias that is better aligned with the detection task, reducing structural noise and mitigating overfitting under limited vulnerability annotations.

In fact, the representation mismatch of CodeBERT often leads to characteristic errors in sequence models. For instance, in a common "false positive" scenario, a function contains a safety-sensitive API (e.g., *memcpy()*/*strcpy()*) but includes a boundary check or explicit validation prior to the call. In such cases, a sequence model may over-rely on the presence of "dangerous tokens" and incorrectly predict a vulnerability, even though the dependency structure indicates the operation is properly constrained. Conversely, VDDA preserves the critical "guard condition → dangerous call" relationship within the optimized CPG and sliced code blocks, guiding the BLSTM-Attention to focus on context that is consistent with the dependency structure rather than superficial lexical cues.

Furthermore, from a security auditing perspective, the predictions of large pre-trained models often lack verifiable, dependency-level evidence chains for analysts. When employing generative LLMs to produce explanations or patch suggestions, there is a risk of 'plausible but incorrect' reasoning (i.e., hallucination), which increases the cost of manual review. Therefore, for the proposed graph-guided vulnerability detection pipeline, the lightweight BLSTM-Attention operating on structured code blocks is superior to fine-tuning sequence-based pre-trained models like CodeBERT in terms of both efficiency and structural-semantic alignment.

4.7 Overall Performance of VDDA

Table 1 presents the experimental results comparing VDDA against eight existing software vulnerability detection tools and models. For a comprehensive evaluation, we categorize the baselines into two groups: four widely adopted rule-based tools (CodeQL [39], FlawFinder [40], SonarQube [41], and RATS [42]), and four representative deep learning-based methods (SySeVR [8], VulBERTa [21], FUNDED [11], and VulDetector [12]). To validate the generalization capability of VDDA, we conduct experiments on both the *N&S* and *D&S* datasets.

Table 1. Performance comparisons between VDDA and eight existing vulnerability detection methods

Models	Accuracy		Precision		Recall		F_1-Score	
	N&S	D&S	N&S	D&S	N&S	D&S	N&S	D&S
CodeQL	0.603	0.601	0.550	0.493	0.723	0.711	0.624	0.582
FlawFinder	0.654	0.642	0.490	0.482	0.516	0.691	0.503	0.562
SonarQube	0.705	0.700	0.555	0.549	0.756	0.745	0.640	0.632
RATS	0.706	0.701	0.569	0.543	0.538	0.802	0.553	0.651
SySeVR	0.852	0.841	0.772	0.742	0.797	0.821	0.784	0.782
VulBERTa	0.873	0.862	0.821	0.815	0.876	0.882	0.848	0.847
FUNDED	0.894	0.889	0.845	0.821	0.891	0.889	0.867	0.854
VulDetector	0.918	0.907	0.875	0.866	0.901	0.893	0.888	0.879
VDDA	0.971	0.941	0.921	0.903	0.958	0.972	0.939	0.936

The rule-based tools, including CodeQL, FlawFinder, SonarQube, and RATS, lack deep modeling of the syntactic and semantic structures inherent in source code. CodeQL identifies code patterns through static analysis based on logical queries, whereas SonarQube relies on predefined rule sets to scan for known issues. In contrast, FlawFinder and RATS perform simple text pattern matching against problematic functions stored in a database. Due to their reliance on rigid, rule-based detection mechanisms, these tools tend to perform poorly on previously unknown or complex vulnerabilities, resulting in the lowest performance among all counterparts.

Among the learning-based approaches, SySeVR and VulBERTa process source code as token sequences. Experimental results show that SySeVR significantly outperforms the rule-based tools. However, by treating code solely as linear token sequences, SySeVR fails to capture the rich syntactic and semantic information embedded in the code structure, leading to inferior performance compared to graph-based methods like FUNDED, VulDetector, and VDDA. VulBERTa employs a language-aware tokenization pipeline and a BERT-like architecture to learn contextual features. While its performance surpasses traditional rule-based tools, it exhibits two notable limitations: First, by modeling code purely as sequences, it fails to explicitly incorporate abstract syntax trees (ASTs) or dependency graphs (CFG/CDG), making it prone to missing critical semantic features in structurally complex code. Second, as a typical black-box model, VulBERTa lacks interpretability, which is a significant drawback for security auditing and code review.

The remaining methods, FUNDED, VulDetector, and VDDA, adopt graph-based representations. FUNDED utilizes an Augmented AST that integrates control and data flows. VulDetector proposes a Weighted Flow Graph (WFG) that incorporates control and data dependencies, assigning different levels of vulnerability sensitivity to nodes to focus on potential vulnerabilities. In contrast, the proposed VDDA focuses on the optimization and processing of Code Property Graphs (CPGs). As evidenced by the results in Table 1, VDDA achieves superior performance compared to both FUNDED and VulDetector, demonstrating the effectiveness of our approach in leveraging structured code representations for vulnerability detection.

5 Conclusion and Future Works

In this study, we propose VDDA, an effective deep learning-based model for software vulnerability detection that incorporates an attention mechanism. To preserve rich syntactic and semantic information for accurate detection, source code is transformed into CPGs using the Joern code analysis tool. A novel CPG optimization algorithm is introduced to simplify the graph structures prior to feeding them into the underlying deep learning model. Several key enhancements (such as three-directional code slicing, organizing slices into code blocks, and separating function names from variable names during code tokenization) are implemented to improve the quality of code vectorization. Furthermore, an attention mechanism is integrated into the VDDA architecture to enhance its capability in identifying critical vulnerability patterns. Experimental results on two benchmark datasets demonstrate that VDDA achieves superior performance compared to many existing approaches in vulnerability detection. However, the current implementation of VDDA is primarily tailored for C/C++ and relies on heuristics for slice selection, which may limit its coverage for certain vulnerability patterns. Additionally, while optimizations reduce learning overhead, the preprocessing cost for large-scale codebases remains noticeable. For future work, we plan to extend support to multiple programming languages through unified intermediate representations, incorporate inter-procedural analysis to capture long-range dependencies, and optimize scalability via incremental and parallel processing.

Acknowledgments

This study was supported by the University Natural Science Research Project of Anhui Province (China) (No. 2024AH050617), the Anhui Provincial Department of Education Teaching Research Project (China) (No. 2023JYXM0883), and the Natural Science Foundation of Anhui Xinhua University (China) (No. 2023ZR001).

References

[1] B. Yu, Y. Fang, Q. Yang, Y. Tang, L. Liu, A survey of malware behavior description and analysis, Frontiers of Information Technology & Electronic Engineering 19(2018) 583-603.

[2] Common Vulnerabilities & Exposures, Published CVE Records. <https://www.cve.org/ProgramOrganization/Board>, 2024 (accessed 03.25.2025).

[3] E.-Z. Zhu, P. Wen, K.-Q. Ni, R.-H. Ma, Implementation of an effective dynamic concolic execution framework for analyzing binary programs, Computers & Security 86(2019) 1-27.

[4] Q.-C. Zou, T. Zhang, R.-P. Wu, J.-X. Ma, M.-C. Li, C. Chen, C.-Y. Hou, From automation to intelligence: Survey of research on vulnerability discovery techniques, Journal of Tsinghua University (Science and Technology) 58(12)(2018) 1079-1094.

[5] E.-Z. Zhu, Q.-X. Yuan, Z.-L. Chen, X.-J. Li, X.-Y. Fang, CCBLA: A lightweight phishing detection model based on CNN, BiLSTM and attention mechanism, Cognitive Computation 15(4)(2023) 1320-1333.

[6] S. Shakya, A. Mukherjee, R. Halder, A. Maiti, A. Chaturvedi, SmartMixModel: Machine learning-based vulnerability detection of Solidity smart contracts, in: Proc. IEEE International Conference on Blockchain, 2022.

[7] G.-J. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: A survey, Proceedings of the IEEE 108(10)(2020) 1825-1848.

[8] Z. Li, D.-Q. Zou, S.-H. Xu, H. Jin, Y.-W. Zhu, Z.-X. Chen, SySeVR: A framework for using deep learning to detect software vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19(4)(2022) 2244-2258.

[9] R.Z.L. Foulefack, A. Marchetto, A rapid review on graph-based learning vulnerability detection, in: Proc. International Conference on the Quality of Information and Communications Technology, 2024.

[10] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: Proc. IEEE Symposium on Security and Privacy, 2014.

[11] H.-T. Wang, G.-X. Ye, Z.-Y. Tang, S.-H. Tan, S.-F. Huang, D.-Y. Fang, Y.-S. Feng, L.-Z. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Transactions on Information Forensics and Security 16(2021) 1943-1958.

[12] L. Cui, Z.-Y Hao, Y. Jiao, H.-Q Fei, X.-C Yun, VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison, IEEE Transactions on Information Forensics and Security 16(2021) 2004-2017.

[13] Z.-Q. Sui, S. Wang, N. Lu, W.-B. Shi, Z.-Q. Liu, MDVul: A semantic-based complex dependency code vulnerability detection using fusion path, Information Fusion 125(2025) 103475.

[14] Joern, The bug hunter's workbench. <https://joern.io>, 2020 (accessed 03.15.2025).

[15] J.-Q. Chang, Z.-J. Ma, B.-H. Cao, E.-Z. Zhu, VDDA: An Effective Software Vulnerability Detection Model Based on Deep Learning and Attention Mechanism, in: Proc. IEEE 26th International Conference on Computer Supported Cooperative Work in Design, 2023.

[16] Y. Mao, Y. Li, J.-T. Sun, Y.-X. Chen, Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks, in: Proc. IEEE International Conference on Big Data, 2020.

[17] B. Zhang, Y.-S. Gao, B.-Y. Kuang, C.-L. Yu, A.-M. Fu, W. Susilo, A Survey on Advanced Persistent Threat Detection: A Unified Framework, Challenges, and Countermeasures, ACM Computing Surveys 57(3)(2025) 62:1-62:36.

[18] N. S. Harzevili, A. B. Belle, J. Wang, S. Wang, Z.-M. Jiang, N. Nagappan, A systematic literature review on automated software vulnerability detection using machine learning, ACM Computing Surveys 57(3)(2024) 1-36.

[19] J.-Y. Ye, Z. Li, X. Tang, D.-Q. Zou, S.-H. Xu, W.-Z. Qiang, H. Jin, A Causal Learning Framework for Enhancing Robustness of Source Code Models, in: Proc. ACM on Software Engineering, 2025.

[20] N. Risse, M. Bohme, Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection, in: Proc. 33rd USENIX Security Symposium, 2024.

[21] H. Hanif, S. Maeis, VulBERTa: Simplified source code pre-training for vulnerability detection, in: Proc. International Joint Conference on Neural Networks, 2022.

[22] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning-based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48(9)(2021) 3280-3296.

[23] D.-Q. Zou, S.-J. Wang, S.-H. Xu, Z. Li, H. Jin, VulDeePecker: A deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing 18(5)(2021) 2224-2236.

[24] Z. Li, D.-Q. Zou, S.-H. Xu, Z.-X. Chen, Y.-W. Zhu, H. Jin, VulDeeLocator: A deep learning-based fine-grained vulnerability detector, IEEE Transactions on Dependable and Secure Computing 19(4)(2022) 2821-2837.

[25] X.-C. Wen, Y.-P. Chen, C.-Y. Gao, H.-Y. Zhang, J.-M. Zhang, Q. Liao, Vulnerability detection with graph simplification and enhanced graph representation learning, in: Proc. IEEE/ACM 45th International Conference on Software Engineering, 2023.

[26] F.-C. Qiu, Z.-X. Liu, X. Hu, X. Xia, G. Chen, X.-Y. Wang, Vulnerability detection via multiple-graph-based code representation, IEEE Transacions on Software Engineering 50(8)(2024) 2178-2199.

[27] S.-C. Cao, X.-B. Sun, X.-X. Wu, D. Lo, L.-L. Bo, B. Li, W. Liu, Coca: Improving and Explaining Graph Neural Network-Based Vulnerability Detection Systems, in: Proc. IEEE/ACM 46th International Conference on Software Engineering, 2024.

[28] Y. Luo, W.-F. Xu, D.-X. Xu, Detecting code vulnerabilities with heterogeneous GNN training, International Journal of Information Security 24(5)(2025) 213.
[29] Y.-M. Wu, D.-Q. Zou, S.-H. Dou, W. Yang, D. Xu, H. Jin, VuICNN: An image-inspired scalable vulnerability detection system, in: Proc. IEEE/ACM 44th International Conference on Software Engineering, 2022.
[30] Z. Sheng, Z.-C. Chen, S.N. Gu, H.-Q. Huang, G.-F. Gu, J. Huang, LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights, ACM Computing Surveys 58(5)(2026) 134:1-134:35.
[31] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, P. Zadorozhny, A. Cheshkov, R. Mussabayev, A. Toleu, G. Tolegen, A. Krassovitskiy, Finetuning large language models for vulnerability detection, IEEE Access 13(2025) 38889-38900.
[32] D. Cao, Y. Liao, X.-W. Shang, RealVul: Can We Detect Vulnerabilities in Web Applications with LLM?, in: Proc. Conference on Empirical Methods in Natural Language Processing, 2024.
[33] J. Lin, D. Mohaisen, From large to mammoth: A comparative evaluation of large language models in vulnerability detection, in: Proc. 2025 Network and Distributed System Security Symposium, 2025.
[34] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, A. Joulin, Advances in pre-training distributed word representations, in: Proc. 11th International Conference on Language Resources and Evaluation, 2018.
[35] National Vulnerability Database. <https://nvd.nist.gov/>, 2024 (accessed 03.28.2025).
[36] Software Assurance Reference Dataset. <https://samate.nist.gov/SARD/test-cases>, 2025 (accessed 04.02.2025).
[37] IBM/D2A. <https://github.com/ibm/D2A>, 2024 (accessed 25.4.10).
[38] C.D. Sestili, W.S. Snavely, N.M. VanHoudnos, Towards security defect prediction with AI. <https://doi.org/10.48550/arXiv.1808.09897>, 2018 (accessed 04.15.2025).
[39] CodeQL. <https://codeql.github.com/>, 2020 (accessed 04.10.2025).
[40] FlawFinder. <http://www.dwh-eeler.com/flawfinder>, 2021 (accessed 25.04.10.2025).
[41] SonarSource, SonarQube – Docker deployment guide, <https://www.sonarsource.com/zh/products/sonarqube/deployment/docker/>, 2025 (accessed 03.15.2025).
[42] Rough audit tool for security. <https://code.google.com/archive/p/rough-auditing-tool-for-security/>, 2022 (accessed 03.15.2025).