Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.081155

ARTICLE

VulSCP: Automated Code Vulnerability Detection via Sequential Convolution and Parallel Attention Mechanism

Zhe Wang[1], Yu Yan[2], Junqi Tong[1], Yijun Lin[1], Dechun Yin[1,*] and Xiaoliang Zhao[1]

[1]School of Information and Network Security, People's Public Security University of China, Beijing, China
[2]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

*Corresponding Author: Dechun Yin. Email: yindechun@ppsuc.edu.cn

Received: 24 February 2026; Accepted: 16 April 2026; Published: 15 June 2026

ABSTRACT: As software applications grow increasingly large and complex, traditional code vulnerability detection methods struggle with performance and efficiency. Although code visualization-based algorithms have demonstrated effectiveness in capturing sparse features and complex workflows in large-scale source code, their capacity to extract global semantic information and intricate long-range dependencies remains limited. Recent large language model (LLM)-based approaches have shown promising accuracy by leveraging rich contextual information, but their high computational cost often limits practical efficiency. To address these challenges, we propose VulSCP, a new framework that integrates sequential convolution with a parallel attention mechanism. Specifically, VulSCP first constructs a semantically weighted graph from the source code, then employs sequential convolution to extract local vulnerability-related features, and finally enhances the global feature representation through parallel attention. Experimental results on large-scale C/C++ function-level datasets show that VulSCP achieves an accuracy of 85.14% and a false positive rate of 17.25%, outperforming the best baseline in accuracy by 1.73 percentage points and reducing the false positive rate by 3.38 percentage points. Moreover, while maintaining high detection accuracy, VulSCP achieves a low average inference time of 1.89 s per sample, showing favorable efficiency compared with the evaluated LLM-based methods. These results suggest that VulSCP is a promising approach for vulnerability detection in large and complex software systems, offering a favorable balance between accuracy and efficiency. The source code of VulSCP is publicly available at https://github.com/Hwzx-ZeL/VulSCP.

KEYWORDS: Code vulnerability detection; sequential convolution; parallel attention; program dependency graph; semantic graph representation

1 Introduction

Recently, the increasing frequency of online software attacks has posed significant challenges to cybersecurity, and this situation is further compounded by the exponential growth in the scale and complexity of modern software systems. Statistics from the National Vulnerability Database (NVD) [1] indicate sustained expansion: disclosures grew by about 39% year-over-year in 2024 relative to 2023 and have more than doubled since 2020 (approximately 2.18×; about 21% compound annual growth over 2020–2024). Early 2025 levels remain elevated, suggesting persistent upward pressure. An industry analysis [2] highlights a similar upward trend observed in practice. These dynamics not only reflect an accelerating pace of vulnerability disclosures but also underscore the increasing technical complexity of these threats. Consequently, an automated detection method that can effectively balance accuracy and computational efficiency is urgently needed, particularly in the context of large-scale software systems.

Traditional code vulnerability detection techniques are generally classified into two main categories: code similarity-based detection [3–5] and pattern-based detection [6–8]. Similarity-based methods are primarily designed to identify vulnerabilities introduced through code cloning. However, these methods often struggle to detect non-clone vulnerabilities, resulting in a high false negative rate. Pattern-based approaches rely on manually defined rules and feature matching to identify vulnerable code. However, due to the inherent limitations in rule formulation, these methods tend to exhibit elevated false positive rates. Deep learning–based techniques [6,9] have emerged as a leading paradigm in static code analysis. By automatically learning vulnerability patterns from large-scale code corpora, these approaches offer a promising solution for achieving both improved detection accuracy and scalability.

Among existing deep learning–based vulnerability detection methods, those that extract program semantics from source code and represent them as graphs, followed by graph-based analysis (e.g., Graph Neural Networks, GNNs), have demonstrated strong detection performance [10–12]. These methods effectively integrate multiple code representations (e.g., AST, NSC, CFG, DFG) and achieve precise vulnerability analysis by modeling the graph structure of source code. However, graph neural network (GNN)–based detection approaches still face significant challenges. On the one hand, the computational complexity of GNNs is substantially higher than that of conventional CNNs, leading to considerable resource consumption when processing large-scale codebases. On the other hand, the extraction of multiple code representations introduces substantial time overhead, which adversely impacts detection efficiency. To balance detection effectiveness and efficiency, we adopt sequential convolution to extract vulnerability-relevant features from a centrality-weighted semantic graph, rather than directly performing costly GNN-based reasoning on explicitly constructed code graphs. Specifically, source code is first embedded into a program dependence graph (PDG) that preserves both control-flow and data-flow dependencies, and the relative importance of code lines is further quantified from multiple complementary perspectives using three centrality measures. Nevertheless, the inherently local receptive field of sequential convolution limits its ability to model global code semantics and long-range dependencies. To better compensate for this limitation under efficiency constraints, we introduce a Parallel Attention (PA) mechanism [13]. Prior CNN-attention-based vulnerability detection models, such as CodeGATNet [14] and the hybrid QCNN-based framework [15], also employ attention to enhance feature modeling, but they differ from our method in both the underlying feature representation and the way attention is incorporated. In our framework, the centrality-weighted graph representation and PA play complementary roles: the former injects structural semantic priors into the input, while the latter dynamically reweights the resulting convolutional features to further emphasize vulnerability-relevant regions and long-range dependencies. Through grouped channel-spatial reweighting performed in parallel, PA improves global semantic perception while preserving the lightweight and efficient characteristics of the overall framework.

In recent years, large language models (LLMs) have demonstrated remarkable potential in the field of code vulnerability detection, attracting significant attention from both academia and industry. For example, the VulLLM framework [16] adopts a Multi-Task Instruction Fine-Tuning strategy that integrates vulnerability localization and explanation tasks, enabling the model to capture the root-cause characteristics of vulnerabilities rather than relying solely on superficial correlations. Similarly, the MSVID (Multitask Self-Instructed Fine-Tuning) method proposed by the CMU team [17] integrates graph neural networks (GNNs) with LLMs, further enhancing the recognition of complex vulnerability patterns. However, these LLM-based detection methods also face notable challenges. First, their performance degrades in complex vulnerability scenarios. Sejfia et al. [18] observed that detection accuracy drops significantly when vulnerabilities span multiple code units. Second, current approaches still rely heavily on lightweight LLMs. Applying LLMs with billions of parameters to vulnerability detection leads to extremely high computational resource demands

and runtime overhead. Such high resource consumption and low efficiency hinder the practical adoption of large-scale LLMs (with over one billion parameters) in industrial vulnerability detection, where both effectiveness and efficiency are critical. Therefore, reducing resource consumption and improving efficiency while maintaining the detection performance of LLMs has become an urgent problem to be solved.

To address the limitations of existing approaches in code vulnerability detection, particularly their difficulty in capturing global semantic features and long-range dependencies, we propose VulSCP, a new and effective detection framework that integrates sequential convolution with a parallel attention mechanism. VulSCP begins by constructing a semantically weighted graph representation of source code, leveraging both control and data dependencies to preserve program semantics. Sequential convolution is then applied to capture local structural patterns associated with vulnerabilities. To further enhance global feature representation, VulSCP incorporates a parallel attention module, consisting of both channel-wise and spatial attention components, enabling the model to effectively emphasize critical features across different semantic dimensions.

Extensive experiments on large-scale C/C++ function-level vulnerability datasets support the effectiveness of VulSCP. The results indicate that the proposed framework performs favorably relative to recent and representative baselines, while maintaining a good balance between semantic modeling capability and computational efficiency. These observations highlight the potential of VulSCP for efficient vulnerability detection in large-scale software settings. The major contributions of this work are as follows:

- **Semantic Graph Representation with Centrality:** To efficiently capture sparse features and complex workflows in large-scale source code, we normalize the source code and construct graph-based representations. By incorporating code centrality metrics, we generate semantically weighted graphs that preserve both data-flow and control-flow semantics.
- **PA-Enhanced Global Semantic Modeling:** To address the limited ability of existing models to capture global semantics and complex logical dependencies, we integrate a parallel attention mechanism. This design balances computational efficiency with enhanced representation of global features.
- **Improved Accuracy and Efficiency:** Extensive experiments demonstrate that VulSCP achieves lower false positive rates and higher accuracy than recent and representative vulnerability detection baselines. Moreover, VulSCP maintains a low detection latency of 1.89 s per code sample, offering favorable efficiency compared with large-model-based detection methods.

2 Related Works

Traditional vulnerability detection methods. Flawfinder [19] performs lexical analysis using a built-in database of known insecure functions and evaluates risks based on context, but it does not conduct data flow or control flow analysis. Cppcheck [20] tokenizes C/C++ source files and matches token sequences to detect vulnerabilities, enabling it to handle complex code structures. FindBugs [21] analyzes Java bytecode via static analysis and examines data flows in the code to identify potential vulnerabilities. Clang Static Analyzer [22] simulates code execution paths using static analysis to detect flaws. Micro Focus Fortify [23] combines static, dynamic, and runtime analyses to uncover security vulnerabilities in software. Checkmarx [24] relies on expert-defined rules to identify code issues. RATS [25] performs static analysis to detect security concerns in source code. Traditional vulnerability detection methods primarily rely on predefined rules and employ pattern matching and lexical analysis. However, these approaches often lack deep semantic understanding and are heavily dependent on manually crafted rules, making them prone to missing complex security issues and resulting in suboptimal detection performance.

Deep learning–based vulnerability detection methods. In recent years, deep learning has led to notable advances in vulnerability detection. Li et al. [6] proposed a slice-level detection model based on

bidirectional long short-term memory networks (BLSTMs), utilizing program slicing for vulnerability identification. Russell et al. [26] extracted code features and trained CNNs and RNNs for function-level detection. Duan et al. [27] introduced a detection model that combines attention mechanisms with BLSTMs and uses tensor-based feature encoding. Feng et al. [28] developed a tool that extracts AST sequences and trains a gated recurrent unit (GRU) model. TokenCNN [26] converts source code into vector representations and applies CNNs for detection. VulDeePecker [6] extracts code slices and uses BLSTM models for vulnerability identification. SySeVR [8] generates slices based on control and data flow information and applies BLSTM for detection. VulDeeLocator [29] compiles source code into LLVM Intermediate Representation (IR), extracts IR slices, and applies BiLSTM for detection. Devign [10] constructs a rich graph representation capturing comprehensive program semantics via advanced program analysis, and utilizes generic GNNs for detection.

LLM-based vulnerability detection methods. LLM-based vulnerability detection methods can be broadly classified into two categories: prompt engineering–based approaches and fine-tuning–based domain adaptation methods. Prompt engineering methods leverage pre-trained large language models (LLMs), such as GPT-3.5 and Code Llama, to perform vulnerability detection without modifying model parameters. Several prompt strategies have been proposed in existing research: clearly defining task goals and roles to enhance model comprehension [30–32]; embedding common vulnerability types (e.g., CWE Top 25) or example code to help models capture key features [30,33]; integrating code dependency information (e.g., from static data flow analysis) into prompts to compensate for LLMs' limited semantic understanding [33,34]; and employing chain-of-thought reasoning to improve detection of complex vulnerabilities [35,36]. While these methods benefit from zero-shot or few-shot settings, their performance is highly sensitive to prompt design (e.g., selection of few-shot examples) [35], and their detection results can lack stability. Fine-tuning methods, by contrast, adapt pre-trained LLMs through supervised learning to specialize in vulnerability detection. Typically, these models are trained end-to-end on datasets such as NVD or SARD to optimize the recognition of vulnerability patterns. To mitigate class imbalance, studies employ data-level resampling [37] and cost-sensitive optimization via loss reweighting [38] to strengthen learning on rare vulnerable samples. To address label noise and incomplete annotations, positive–unlabeled learning reframes training with positives and unlabeled examples, often combined with selective pseudo-labeling and mixed-supervision objectives [39]. In parallel, incorporating structured program information (e.g., PDGs/CFGs) further enhances code semantics [40–42]. Some approaches also combine GNNs or BLSTM architectures to build multimodal models that simultaneously capture code sequences and structural features [43–45]. While fine-tuning offers more stable accuracy, it entails high computational costs and prolonged training time, making it less feasible for efficient deployment in industrial-scale vulnerability detection.

3 Method

Existing methods for vulnerability detection often face a trade-off between computational efficiency and semantic modeling capability when applied to large-scale codebases. To address this issue, VulSCP is designed as a lightweight framework that combines semantically weighted graph representation, sequential convolution, and parallel attention, as illustrated in Fig. 1.

Specifically, the framework first extracts the Program Dependency Graph (PDG) from source code and performs code embedding. Based on the PDG, it then constructs a multi-channel semantically weighted representation using three complementary centrality measures, namely degree centrality, Katz centrality, and closeness centrality, so as to preserve both structural dependency information and the relative importance of code lines. On this basis, sequential convolution is employed to capture local vulnerability-related patterns,

while the parallel attention module further enhances global semantic perception and long-range dependency modeling. Finally, the resulting features are fused and fed into the classifier to perform end-to-end vulnerability detection.

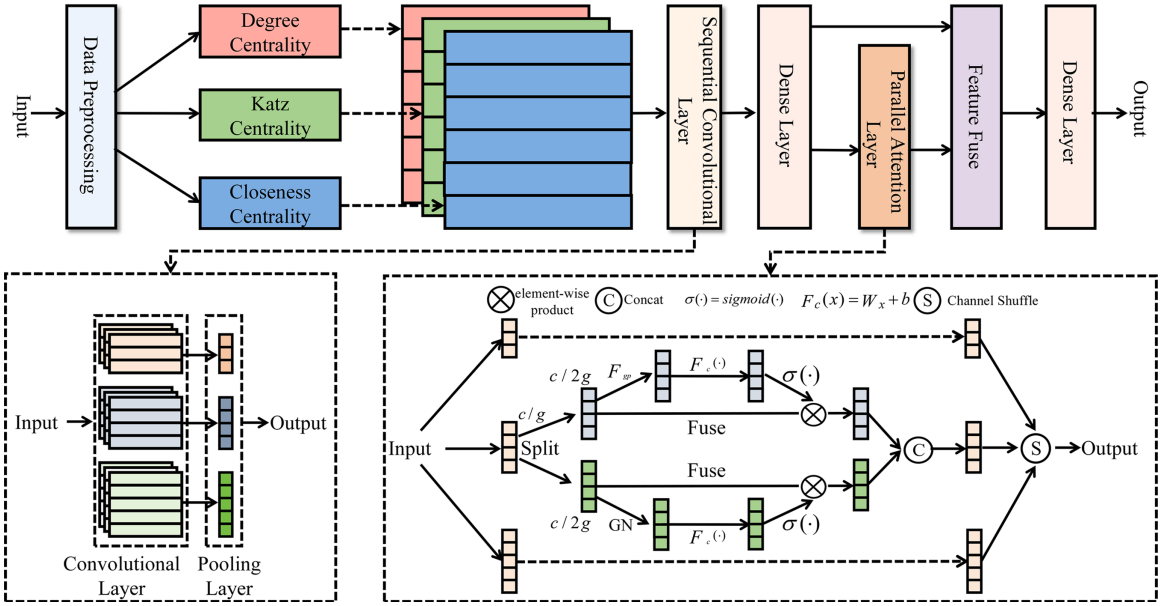


Figure 1: An overview of vulnerability detection via sequential convolution and a parallel attention mechanism.

3.1 Code Representation

3.1.1 Code Normalization

To achieve both accuracy and scalability in vulnerability detection, this paper adopts static analysis techniques to extract program semantics from source code and represent them in graph form. Before generating the graph representation of functions, the source code undergoes a normalization process to ensure consistency and robustness against common code transformations while preserving program semantics. This normalization is performed in three steps:

Fig. 2 illustrates the detailed transformation of a function across different levels of normalization. The normalization process includes the following operations:

- Removal of comments: All comments that do not affect the program's semantics are eliminated.
- Standardization of variable names: User-defined variable names are replaced with ordered, standardized placeholders (e.g., VAR1).
- Standardization of function names: User-defined function names are replaced with ordered, standardized identifiers (e.g., FUN1).

This normalization strategy ensures that superficial variations in code structure do not interfere with the vulnerability detection process, allowing the model to focus on meaningful semantic patterns.

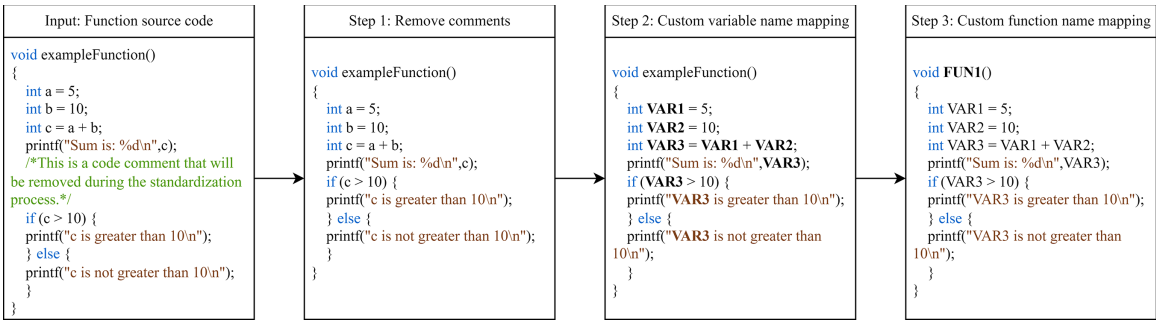


Figure 2: Code normalization.

3.1.2 Code Graph Representation

The embedded representation of code is primarily constructed through Program Dependency Graph (PDG) extraction and sentence embedding techniques (specifically, sent2vec), enabling the transformation of source code into graph-based representations.

PDG is a widely adopted method for modeling code structure, capturing both data flow and control flow dependencies within the source code. In this work, we utilize Joern, an open-source code analysis platform for C/C++, to extract PDGs from the source code. Sent2vec is a robust and efficient sentence embedding technique that leverages a simple unsupervised objective to learn distributed representations of sentences. By applying sent2vec, each line of code can be encoded into a fixed-length vector, treating the code line as a natural language sentence. Specifically, each statement in the source code is mapped to a corresponding node in the PDG. Each line is treated as a sentence and embedded using the sent2vec model, producing a dense vector representation. This process integrates semantic information into the graph structure, facilitating downstream learning tasks. Fig. 3 illustrates the overall graph construction pipeline. In the figure, red edges and blue edges denote data flow and control flow dependencies, respectively, between different lines of code within the function.

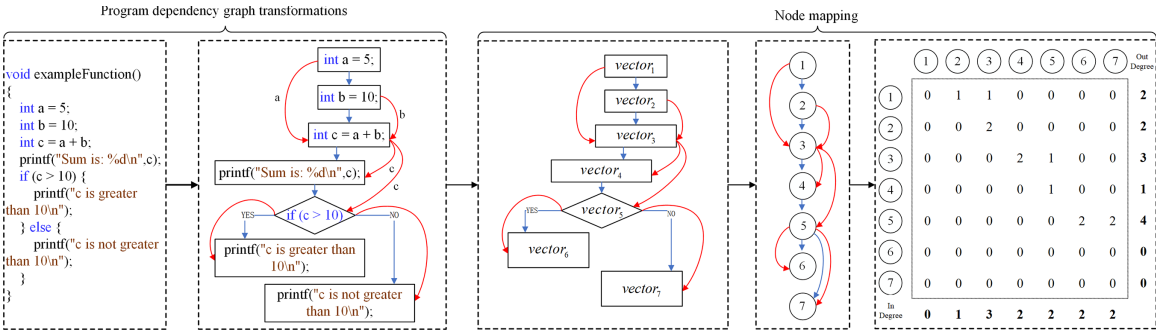


Figure 3: Code embedding representation.

3.2 Code Centrality Representation

To characterize the relative importance of different code lines in the program semantic structure, we introduce centrality measures on the Program Dependency Graph (PDG) and perform structural importance weighting for code slices. A PDG is a graph in which nodes correspond to statements and edges represent control and data dependencies; therefore, centrality theory from graph theory and social network analysis provides a principled way to quantify the structural roles of statements.

Unlike social networks where node attributes and relationship semantics can be open-ended and dynamically evolving, program semantics are more deterministic: statement boundaries, dependencies, and execution logic are explicit, and the structural properties of PDG nodes are accordingly well defined. In this setting, overly complex metrics that rely on rich node attributes or high-order statistics may introduce noise and additional computational overhead without yielding more robust representations. Therefore, we adopt three classical and complementary centrality measures—degree, Katz, and closeness centrality—to quantify slice importance, capturing local connectivity, multi-hop influence propagation, and global proximity, respectively, as illustrated in Fig. 4. Accordingly, we compute these three centralities as follows.

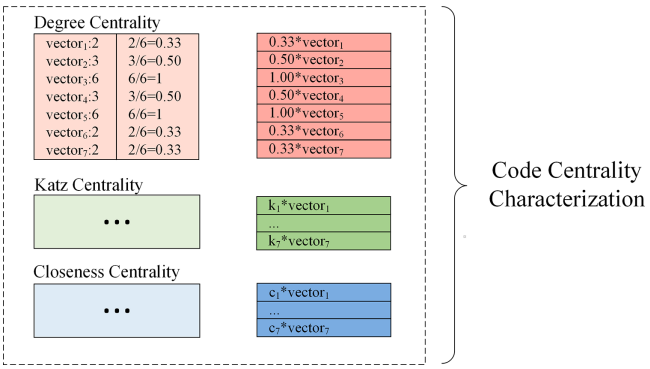


Figure 4: Code centrality representation.

Degree Centrality evaluates the importance of a node based on the number of its direct connections to other nodes. For a given code slice i, the degree centrality x_i^D is defined as:

$$x_i^D = \frac{\deg(i)}{N-1}, \tag{1}$$

where $\deg(i)$ denotes the degree of the node i, and N is the total number of nodes in the graph. A higher x_i^D value indicates that the code line involves more intensive data and control dependencies, suggesting it may play a more central role in the program execution flow.

Katz Centrality measures a node's potential influence by considering not only its immediate neighbors but also the weighted influence of more distant nodes (with attenuation over distance). For a code slice i, the Katz centrality x_i^K is defined as:

$$x_i^K = \alpha \sum_j A_{ij} x_j^K + \beta, \tag{2}$$

where A is the adjacency matrix of the graph G with eigenvalues λ, α is a damping factor, such that $\alpha < \frac{1}{\lambda_{\max}}$. β is a constant that controls the initial centrality.

Closeness Centrality quantifies how close a node is to all other nodes in the graph by computing the average length of the shortest paths from that node to every other node. For a code slice i, the closeness centrality x_i^C is given by:

$$x_i^C = \frac{N-1}{\sum_{j \neq i} d(i,j)}, \tag{3}$$

where $d(i, j)$ is the shortest path distance between nodes i and j, and N is the total number of nodes in the graph. Nodes with high x_i^C reside "centrally" in the PDG, implying roles in system-wide data exchange or control coordination.

Using these three centrality measures, we obtain per-line importance scores from complementary perspectives, and use each score to reweight the corresponding line embedding, yielding three centrality-aware feature channels for downstream learning.

3.3 Sequential Convolution and Parallel Attention Neural Network

3.3.1 Sequential Convolution Layer

To meet the dual requirements of efficiency and accuracy in large-scale vulnerability detection, we design a sequential convolution layer to extract multi-granularity local patterns from code-slice representations. The input to this layer is the centrality-aware feature tensor $X_C^{\text{in}} \in \mathbb{R}^{c_{\text{in}} \times L \times d}$, where c_{in} denotes the number of centrality channels (three in this work), L is the number of code lines in the slice, and d is the embedding dimension of each code line.

We adopt a multi-branch convolution architecture with M parallel branches. The m-th branch applies a convolution operator $\text{Conv}_{k_m}(\cdot)$ whose kernel height k_m controls the receptive field along the code-line (sequence) dimension, while the kernel spans the full embedding dimension. Therefore, the convolution only slides along the sequence dimension, enabling efficient sequential feature extraction. The output of the m-th branch is defined as:

$$X_{C,m} = \text{ReLU}\big(\text{Conv}_{k_m}(X_C^{\text{in}})\big), \tag{4}$$

where $\text{ReLU}(\cdot)$ is the activation function. With a valid convolution (stride 1, no padding), the resulting feature map satisfies $X_{C,m} \in \mathbb{R}^{C_m \times (L-k_m+1) \times 1}$, where C_m is the number of output channels (filters) in the m-th branch.

To obtain a fixed-length representation for each branch, we perform global max pooling over the sequence dimension:

$$z_{C,m} = \text{GMP}(X_{C,m}), \tag{5}$$

which yields a compact vector $z_{C,m} \in \mathbb{R}^{C_m}$. The multi-scale features from all branches are then concatenated to form the unified sequential representation:

$$z_C = \text{Concat}(z_{C,1}, z_{C,2}, \ldots, z_{C,M}). \tag{6}$$

Finally, we apply dropout and a linear projection to produce the output of the sequential convolution layer:

$$X_C^{\text{out}} = W_C \cdot \text{Dropout}(z_C) + b_C, \tag{7}$$

where W_C and b_C are the weight matrix and bias vector of the linear projection layer, respectively, and X_C^{out} serves as the shallow sequential feature for subsequent parallel attention and feature fusion.

3.3.2 Parallel Attention Layer

The parallel attention mechanism fuses channel and spatial attention to enhance the deep semantic representation of code sequences, thereby improving the precision of vulnerability detection. We first apply a pointwise linear projection along the channel dimension at each spatial position to obtain attention features. This operation is equivalent to a shared fully connected transformation applied independently at

each position, and is commonly implemented as a 1×1 convolution. We then perform channel and spatial reweighting within each group. The calculation is defined as:

$$X_A^{\text{in}} = W_A \times X_C^{\text{out}} + b_A, \tag{8}$$

where $X_A^{\text{in}} \in \mathbb{R}^{a_{\text{in}} \times H_{\text{out}} \times W_{\text{out}}}$ denotes the projected feature map, W_A, b_A are the trainable weights and biases of the projection, and a_{in} is the number of attention input channels.

The attention input X_A^{in} is divided into G groups along the channel dimension, with each group containing the same number of channels. The g-th group is denoted by $X_{A,g}^{\text{in}}$. Channel and spatial attention are then applied to each group, producing the output $X_{A,g}^{\text{out}}$, as defined by:

$$X_{A,g}^{\text{in}} = [X_{A,g}^{C,\text{in}}, X_{A,g}^{S,\text{in}}], \quad X_{A,g}^{\text{out}} = [X_{A,g}^{C,\text{out}}, X_{A,g}^{S,\text{out}}], \tag{9}$$

where $X_{A,g}^{C,\text{in}}, X_{A,g}^{C,\text{out}} \in \mathbb{R}^{C_{CA} \times H_{\text{out}} \times W_{\text{out}}}$ denote the input and output tensors of channel attention, and $X_{A,g}^{S,\text{in}}, X_{A,g}^{S,\text{out}} \in \mathbb{R}^{C_{SA} \times H_{\text{out}} \times W_{\text{out}}}$ denote the input and output tensors of spatial attention.

The channel and spatial attention computations are defined as:

$$\begin{aligned} X_{A,g}^{C,\text{out}} &= \sigma\left(W_{CA} \times \text{Pool}\left(X_{A,g}^{C,\text{in}}\right) + b_{CA}\right) \cdot X_{A,g}^{C,\text{in}}, \\ X_{A,g}^{S,\text{out}} &= \sigma\left(W_{SA} \times \text{GN}\left(X_{A,g}^{S,\text{in}}\right) + b_{SA}\right) \cdot X_{A,g}^{S,\text{in}}. \end{aligned} \tag{10}$$

$\text{Pool}(\cdot)$ is a pooling operation, $\text{GN}(\cdot)$ is a group normalization operation, W_{CA}, b_{CA} are the weights and biases for channel attention, and W_{SA}, b_{SA} are the weights and biases for spatial attention. $\sigma(\cdot)$ denotes the Sigmoid activation function.

Once attention is applied in both channel and spatial dimensions, the resulting feature maps are fused to generate final attention-enhanced feature representations, thereby improving interaction between dimensions.

$$X_A^{\text{out}} = \text{Shuffle}\left(\left[X_{A,0}^{\text{out}}, \ldots, X_{A,g}^{\text{out}}, \ldots, X_{A,G-1}^{\text{out}}\right]\right). \tag{11}$$

$X_A^{\text{out}} \in \mathbb{R}^{a_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$, where a_{out} denotes the number of output attention channels. The function $\text{Shuffle}(\cdot)$ indicates a shuffling and reorganization operation over the group-wise outputs $X_{A,g}$.

3.3.3 Feature Fusion and Output Layer

After extracting features through sequential convolution and parallel attention, we obtain the shallow code features X_C^{out} from the sequential convolution and the deep semantic features X_A^{out} from the parallel attention. These are then fused and passed through a fully connected classification layer and a Softmax function to obtain the final classification result:

$$\begin{aligned} X &= \text{Concat}\left(X_A^{\text{out}}, X_C^{\text{out}}\right), \\ \hat{y} &= \text{Softmax}\left(W_F \times X + b_F\right). \end{aligned} \tag{12}$$

$\text{Concat}(\cdot, \cdot)$ denotes the concatenation of feature tensors along the channel dimension. Before the fully connected classification layer, we apply pooling and flattening to the fused feature map. W_F and b_F are the weight matrix and bias vector of the fully connected layer, respectively.

4 Experimental Settings

4.1 Dataset

This study evaluates VulSCP on C/C++ function-level vulnerability detection. Our main dataset is constructed from three sources:

- National Vulnerability Database (NVD) [1]: 1384 vulnerable C/C++ function samples.
- Software Assurance Reference Dataset (SARD) [46]: 12,303 vulnerable and 21,057 non-vulnerable C/C++ function samples.
- BigVul (non-vulnerable set) [47]: 5913 non-vulnerable C/C++ function samples randomly sampled from the BigVul corpus.

We directly use the binary labels provided by the corresponding sources (vulnerable vs. non-vulnerable) without manual relabeling.

During sample collection, we first normalize code (e.g., comment removal and lexical normalization) to improve parsing robustness, and then check whether the normalized function is an exact duplicate of a previously collected sample. If an exact duplicate is found, it is discarded immediately; therefore, the sample counts reported above correspond to the final dataset used in our experiments. To mitigate data leakage, each function sample is assigned to exactly one split (training/validation/test), and all model selection is performed on the validation split only.

To examine cross-dataset generalization, we additionally evaluate on the ReVeal benchmark dataset for vulnerable function detection [48]. A summary of the dataset statistics is provided in Table 1.

Table 1: Dataset statistics for vulnerability detection experiments.

Dataset	Samples	Vul	Non-vul	Vul Ratio (%)
NVD+SARD+BigVul	40,657	13,687	26,970	33.66
ReVeal	18,169	1664	16,505	9.16

4.2 Evaluation Metric

The following metrics are used to evaluate the model's performance on vulnerability detection tasks:

- Accuracy: The proportion of correctly predicted samples among all samples. Higher accuracy indicates that the model's predictions are closer to the ground truth.
- False Positive Rate (FPR): The proportion of negative samples incorrectly classified as positive. A lower FPR reflects better performance in identifying non-vulnerable code.
- False Negative Rate (FNR): The proportion of positive samples incorrectly classified as negative. A lower FNR implies the model is more effective at detecting true vulnerabilities.
- F1 Score: The harmonic mean of precision and recall provides a balanced evaluation of the model's overall performance.
- Runtime Overhead: The time taken by the model to perform vulnerability detection. Lower runtime indicates higher efficiency and better scalability for large-scale code analysis.

4.3 Experimental Setup

We use a random function-level split of the dataset into training, validation, and test sets with a ratio of 7:2:1, rather than adopting a project-wise split. To reduce the effect of training randomness, we keep the data split fixed and repeat the training and evaluation process five times with different random seeds.

Unless otherwise specified, all results in this paper are reported as mean ± standard deviation over the five independent runs under the same experimental setting.

The training batch size is set to 32, and the model is trained for 300 epochs with a learning rate of 0.001. The AdamW optimizer is used along with a cross-entropy loss function. In the sequential convolution module, we use 10 parallel convolution branches with kernel heights ranging from 1 to 10 to capture multi-scale local patterns, and set the dropout rate to 0.1 for regularization. The output hidden dimension of the sequential convolution module is set to 128 with a sequence width Wc_{out} of 100. The parallel attention module is configured with 64 input channels a_{in}, 8 parallel groups G, and an output dimension a_{out} of 512. Joern is used to extract program dependency graphs from the source code, networkx is applied to compute graph centrality metrics, and sent2vec is employed to generate embeddings for code segments.

The key hardware and software configurations of the experimental environment include a 12th Gen Intel(R) Core(TM) i9-13900H 2.60 GHz CPU, an NVIDIA GeForce RTX 4060 GPU, 32 GB of RAM, and Ubuntu 22.

For a reproducible runtime comparison with the prompt-based LLM baselines, we specify their deployment environment and end-to-end timing protocol. These methods perform vulnerability detection by sending prompts to deployed large language models through an OpenAI-compatible API. The models were served on a dedicated inference server with 8× NVIDIA H20 96 GB GPUs, an Intel Xeon Platinum 8480 CPU, 64 vCPUs, 768 GB RAM, and a 4 TB NVMe SSD, running Ubuntu Server 22.04.5 LTS, NVIDIA Driver 550.54.14, Docker Engine 28.5.1, and NVIDIA Container Toolkit 1.18.0. Inference was deployed via vLLM v0.18.0. During evaluation, prompt requests were sent from a local client machine to the server, and the reported runtime includes the end-to-end overhead under this deployment setting.

4.4 Baselines

The baseline methods in this study span a variety of vulnerability detection techniques, including traditional rule-based tools, deep learning-based approaches, and cutting-edge large language model (LLM) frameworks. These baselines cover recent and representative developments in vulnerability detection technology.

The rule-based tools include Checkmarx [24], Flawfinder [19], and RATS [25]. Checkmarx identifies vulnerabilities based on expert-defined rules, Flawfinder detects potential security issues using predefined patterns, and RATS performs static analysis to uncover security flaws in source code. These tools scan and analyze code by matching it against pre-established rules and patterns, offering efficient detection of common vulnerabilities.

Deep learning-based approaches include AugSliceVul [49], TokenCNN [26], VulDeePecker [6], SySeVR [8], and Devign [10]. TokenCNN converts source code into vector representations and trains a convolutional neural network (CNN) for vulnerability detection. VulDeePecker extracts code slices and utilizes a bidirectional long short-term memory (BiLSTM) network for classification. SySeVR uses control and data flow to extract program slices and applies a BiLSTM model for detection. Devign performs deep program analysis to extract comprehensive semantic graph representations and employs general-purpose graph neural networks to detect vulnerabilities. These deep learning methods enable the identification of complex code patterns and enhance detection capabilities. AugSliceVul augments the program dependency graph and leverages an optimized CodeBERT encoder to capture richer structural dependencies and semantic context for vulnerability classification.

LLM-based approaches can be classified into two categories. The first involves prompt-based detection, where APIs of cutting-edge LLMs are directly called for zero-shot vulnerability detection. The models used

include the Qwen2.5 series with 14, 32, and 72B parameters, the Qwen3 series with 8, 14, and 32B parameters, the GLM4 series with 9 and 32B parameters, and DeepSeek-v3. The second category includes VulLLM [16], which integrates a multi-task learning framework with data augmentation techniques and applies low-rank adaptation (LoRA) for efficient fine-tuning. VulLLM adapts general LLMs for the vulnerability detection task through domain-specific fine-tuning. In this study, four high-performing VulLLM models are selected as baselines, namely VulLLM-CL-7B, VulLLM-CL-13B, VulLLM-SC-7B, and VulLLM-SC-15B. These are based on the CodeLlama-7B/13B and StarCoder-7B/15B architectures and have been fine-tuned with supervised training specifically for vulnerability detection.

These baselines represent a comprehensive spectrum of methodologies, ranging from traditional rule-based systems to advanced deep learning and LLM-based approaches. Comparative results suggest that VulSCP has advantages in detection accuracy, inference efficiency, and the modeling of global semantic features and complex logical dependencies in source code.

5 Experimental Results and Analysis

5.1 Overall Comparison with Baselines

In this section, we compare VulSCP with a diverse set of baselines using multiple evaluation metrics. In our experimental setting, increasing the window size leads to larger input representations and higher memory usage. Considering the trade-off between performance and resource consumption, the window size is set to 100 code lines. Samples with lengths close to this setting are used in the experiments.

As shown in Table 2, the three rule-based tools, namely Checkmarx, Flawfinder, and RATS, exhibit limited detection performance. Their ACC values remain relatively low, ranging from 48.57% to 60.13%, with relatively high FPR and FNR, indicating that predefined rules and patterns are insufficient to cover diverse vulnerability forms in complex source code. TokenCNN achieves the lowest runtime overhead (0.22 s), but its ACC is only 72.33%, suggesting that a lightweight CNN operating on plain code representations is efficient but insufficient for reliable vulnerability detection.

Table 2: Performance comparison of vulnerability detection methods. All results are reported as mean ± standard deviation over five runs. The best results are **bolded** and the second-best are underlined. Abbreviations: P LLM (prompt-based LLM); FT LLM (fine-tuned LLM).

Type	Technique	FPR (%)	FNR (%)	ACC (%)	Time (s)
P LLM	Qwen2.5-14B	$30.9_{\pm 0.82}$	$26.37_{\pm 0.42}$	$70.77_{\pm 0.55}$	2.75
	Qwen2.5-32B	$30.76_{\pm 0.59}$	$29.16_{\pm 0.48}$	$69.83_{\pm 0.43}$	2.24
	Qwen2.5-72B	$27.32_{\pm 0.55}$	$21.37_{\pm 0.36}$	$74.87_{\pm 0.35}$	2.06
	Qwen3-8B	$25.78_{\pm 0.79}$	$13.96_{\pm 0.45}$	$78.58_{\pm 0.56}$	23.1
	Qwen3-14B	$23.58_{\pm 0.70}$	$9.63_{\pm 0.29}$	$81.56_{\pm 0.51}$	22.13
	Qwen3-32B	$21.37_{\pm 0.59}$	$9.31_{\pm 0.28}$	$83.03_{\pm 0.49}$	30.11
	GLM4-9B	$22.44_{\pm 0.86}$	$18.78_{\pm 0.41}$	$78.91_{\pm 0.43}$	1.95
	GLM4-32B	$27.04_{\pm 0.88}$	$15.05_{\pm 0.42}$	$77.38_{\pm 0.56}$	3.34
	DeepSeek-v3	$21.65_{\pm 0.64}$	$15.57_{\pm 0.46}$	$80.59_{\pm 0.42}$	3.21
FT LLM	VulLLM-CL-7B	$24.98_{\pm 0.71}$	$7.65_{\pm 0.29}$	$81.41_{\pm 0.35}$	9.21
	VulLLM-CL-13B	$22.1_{\pm 0.68}$	$7.98_{\pm 0.27}$	$83.1_{\pm 0.33}$	9.57
	VulLLM-SC-7B	$27.9_{\pm 0.67}$	$\mathbf{6.09_{\pm 0.33}}$	$80.21_{\pm 0.32}$	9.34
	VulLLM-SC-15B	$26.1_{\pm 0.42}$	$\underline{6.78}_{\pm 0.58}$	$81.02_{\pm 0.34}$	9.79

(Continued)

Table 2 (continued)

Type	Technique	FPR (%)	FNR (%)	ACC (%)	Time (s)
Non-LLM	Checkmarx	$27.24_{\pm 0.23}$	$68.9_{\pm 0.79}$	$58.97_{\pm 0.67}$	–
	Flawfinder	$26.71_{\pm 0.20}$	$65.97_{\pm 0.98}$	$60.13_{\pm 0.75}$	–
	RATS	$54.51_{\pm 0.34}$	$45.6_{\pm 0.52}$	$48.57_{\pm 0.29}$	–
	TokenCNN	$27.88_{\pm 0.78}$	$27.67_{\pm 0.58}$	$72.33_{\pm 0.70}$	**0.22**
	Devign	$21.72_{\pm 0.65}$	$8.89_{\pm 0.30}$	$82.63_{\pm 0.33}$	10.41
	SySeVR	$22.89_{\pm 0.59}$	$14.19_{\pm 0.41}$	$80.08_{\pm 0.33}$	7.38
	VulDeePecker	$24.48_{\pm 0.52}$	$17.8_{\pm 0.85}$	$77.75_{\pm 0.60}$	6.95
	AugSliceVul	$\underline{20.63}_{\pm 0.73}$	$9.03_{\pm 0.34}$	$\underline{83.41}_{\pm 0.32}$	9.57
	VulSCP	$\mathbf{17.25_{\pm 0.58}}$	$13.24_{\pm 0.27}$	$\mathbf{85.14_{\pm 0.32}}$	$\underline{1.89}$

VulDeePecker and SySeVR, which are both based on program slicing and bidirectional recurrent modeling, achieve better results than TokenCNN. Compared with VulDeePecker, SySeVR reduces FPR from 24.48% to 22.89% and FNR from 17.80% to 14.19%, while improving ACC from 77.75% to 80.08%. This suggests that incorporating control-flow information is beneficial. However, both methods still underperform VulSCP, whose ACC reaches 85.14%. The gap indicates that slice-based sequential modeling alone remains insufficient for capturing richer semantic dependencies in code.

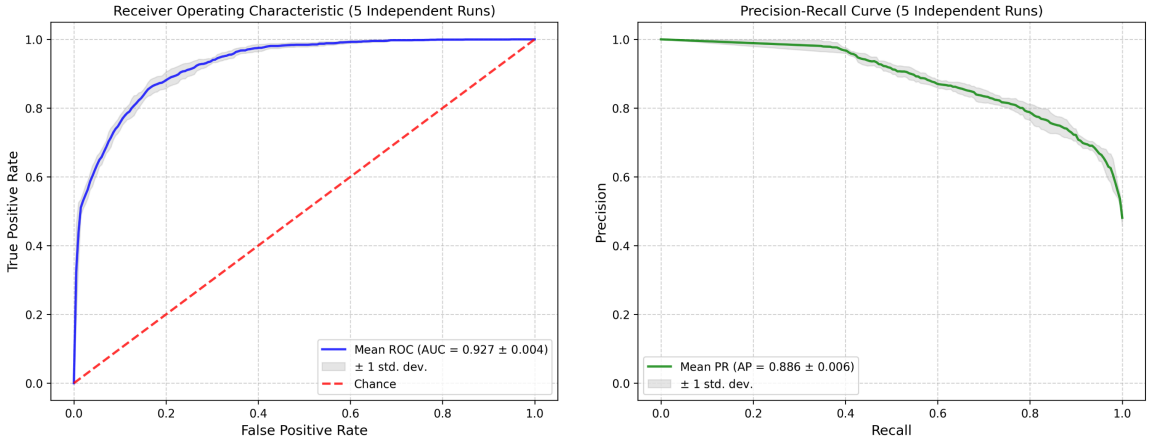
Among the non-LLM deep learning baselines, AugSliceVul [49] is the strongest competitor, achieving 83.41% ACC, 20.63% FPR, and 9.03% FNR. Compared with AugSliceVul, VulSCP improves ACC by 1.73 percentage points and reduces FPR by 3.38 percentage points, while requiring substantially less runtime (1.89 vs. 9.57 s). Although AugSliceVul attains a lower FNR, the overall comparison indicates that VulSCP provides a better balance between accuracy, false-alarm control, and efficiency. Devign, as a GNN-based baseline, also achieves competitive performance with 82.63% ACC and 8.89% FNR, but its runtime reaches 10.41 s, which is more than five times that of VulSCP. This result highlights the advantage of VulSCP in achieving competitive semantic modeling capability without incurring the higher computational overhead of explicit graph neural reasoning.

Among the prompt-based LLM approaches, GLM4-9B and DeepSeek-v3 achieve 78.91% and 80.59% ACC, respectively, with runtime overheads of 1.95 and 3.21 s. The Qwen2.5 series shows runtime relatively close to that of VulSCP, but its detection metrics remain clearly lower. In contrast, the Qwen3 inference models obtain stronger FNR performance, with the best value reduced to 9.31%, indicating improved sensitivity to vulnerable samples. However, this gain is accompanied by substantially higher runtime. Compared with the evaluated LLM-based methods, VulSCP achieves the highest ACC (85.14%) and the lowest FPR (17.25%), while maintaining a low runtime overhead of 1.89 s. These results suggest that VulSCP offers a favorable balance between detection effectiveness and efficiency relative to the evaluated LLM-based methods.

For the fine-tuned LLM-based models (VulLLM series), a similar trade-off can be observed. Although these models further reduce FNR by about 5.2%–7.1% relative to VulSCP, they also exhibit higher FPR, indicating a tendency to over-predict vulnerable samples. In addition, their runtime overhead remains notably higher than that of VulSCP. Therefore, while fine-tuned LLMs show strong sensitivity to vulnerability samples, VulSCP maintains a more balanced error profile together with substantially lower computational cost.

Overall, VulSCP achieves the highest ACC and the lowest FPR among all compared methods, while keeping runtime low. These results show that the proposed framework compares favorably with traditional tools, recent and representative deep learning baselines, and the considered LLM-based approaches.

To further evaluate the classification capability of VulSCP under different decision thresholds, we additionally report the Receiver Operating Characteristic (ROC) curve and the Precision–Recall (PR) curve, as shown in Fig. 5. Based on the statistics of five independent runs, VulSCP achieves a mean AUC of 0.927 ± 0.004 and a mean Average Precision (AP) of 0.886 ± 0.006, indicating strong and stable overall discriminative ability. The ROC curve consistently bends toward the upper-left corner, suggesting that VulSCP can achieve a high true positive rate under a low false positive rate. The PR curve further shows that the model maintains high precision as recall increases, reflecting robust positive-class recognition. Moreover, the narrow standard-deviation bands around both curves demonstrate the stability and robustness of VulSCP across repeated experiments.



a ROC curve. **b** Precision–Recall curve.

Figure 5: ROC and Precision–Recall curves of VulSCP over five independent runs (mean curve with standard-deviation band).

5.2 Cross-Dataset Generalization Analysis

To further assess the generalization ability of VulSCP under distribution shift, we conduct cross-dataset experiments. Specifically, the model is trained on the source-domain dataset NVD+SARD+BigVul and directly tested on the target-domain dataset ReVeal, without any fine-tuning on the target domain. As summarized in Table 1, the two domains differ substantially in both scale and class distribution, making this setting a more challenging evaluation of transferability.

The cross-dataset results are summarized in Table 3.

Among all compared methods, VulSCP achieves the highest ACC (85.55%) and the lowest FPR (9.74%), while maintaining a runtime of only 1.95 s. Compared with the strongest non-LLM baseline AugSliceVul, VulSCP improves ACC from 84.95% to 85.55% and further reduces FPR from 11.02% to 9.74%, with only a slight difference in FNR (55.16% vs. 55.63%). In addition, VulSCP shows the smallest ACC variance ($85.55_{\pm 0.11}$) and a lower FPR variance ($9.74_{\pm 0.21}$) than AugSliceVul ($11.02_{\pm 0.31}$), indicating more stable performance across repeated runs. These results suggest that VulSCP maintains favorable cross-dataset generalization while preserving clear efficiency advantages.

Table 3: Performance comparison on the cross-dataset generalization. All results are reported as mean ± standard deviation over five runs. The best results are **bolded** and the second-best are underlined.

Type	Method	FPR (%)	FNR (%)	ACC (%)	Time (s)
LLM	Qwen3-32B	$39.36_{\pm 0.23}$	$\mathbf{40.16_{\pm 0.44}}$	$60.56_{\pm 0.30}$	41.91
	GLM4-9B	$28.95_{\pm 0.09}$	$56.52_{\pm 0.12}$	$68.37_{\pm 0.15}$	2.01
	DeepSeek-v3	$34.38_{\pm 0.13}$	$\underline{51.86}_{\pm 0.22}$	$64.02_{\pm 0.14}$	3.28
Non-LLM	TokenCNN	$26.8_{\pm 0.38}$	$69.53_{\pm 0.74}$	$69.29_{\pm 0.56}$	**0.24**
	Devign	$30.25_{\pm 0.12}$	$55.68_{\pm 0.55}$	$67.22_{\pm 0.16}$	11.83
	SySeVR	$15.04_{\pm 0.34}$	$77.69_{\pm 0.72}$	$78.67_{\pm 0.24}$	8.2
	VulDeePecker	$11.04_{\pm 0.47}$	$86.24_{\pm 0.40}$	$82.07_{\pm 0.29}$	7.79
	AugSliceVul	$\underline{11.02}_{\pm 0.31}$	$55.16_{\pm 0.27}$	$\underline{84.95}_{\pm 0.23}$	10.46
	VulSCP	$\mathbf{9.74_{\pm 0.21}}$	$55.63_{\pm 0.49}$	$\mathbf{85.55_{\pm 0.11}}$	$\underline{1.95}$

The target domain is highly imbalanced, with only 9.16% vulnerable samples, so ACC alone is insufficient for a complete evaluation. Under this setting, some methods still obtain seemingly competitive ACC but exhibit extremely high FNR. For example, VulDeePecker and SySeVR achieve 82.07% and 78.67% ACC, respectively, yet their FNR values remain as high as 86.24% and 77.69%. This indicates that these models tend to fit the majority class and fail to adequately recognize vulnerable samples under cross-dataset distribution shift. By contrast, VulSCP attains higher ACC together with much lower FPR, indicating a more favorable error trade-off in the target domain.

The LLM-based methods are included here mainly for reference, rather than as strictly comparable cross-dataset transfer baselines, since they are not trained on NVD+SARD+BigVul and then transferred to ReVeal in the same way as the non-LLM models. Under this setting, some LLM-based methods show relatively favorable FNR performance on ReVeal, but their ACC remains clearly lower than that of VulSCP. For example, GLM4-9B achieves 68.37% ACC, while Qwen3-32B and DeepSeek-v3 reach only 60.56% and 64.02% ACC, respectively. This indicates that, although LLM-based methods may be more sensitive to vulnerable samples in this setting, their overall detection performance on ReVeal is still limited compared with VulSCP.

5.3 Ablation Study

5.3.1 Ablation on the Parallel Attention Mechanism

To evaluate the impact of the parallel attention mechanism on the model's ability to capture long-range dependencies, we conducted an ablation study on the attention module. VulSCP is composed of a sequential convolution module and a parallel attention module, the latter of which consists of a channel attention module and a spatial attention module. Accordingly, we designed three variant models for comparative experiments:

- VulSCP: Includes both the sequential convolution layer and the parallel attention layer, which integrates channel and spatial attention modules.
- r.m. Spatial Attention: Includes the sequential convolution layer and only the channel attention module from the parallel attention layer, with the spatial attention module removed.
- r.m. Channel Attention: Includes the sequential convolution layer and only the spatial attention module from the parallel attention layer, with the channel attention module removed.

- r.m. Parallel Attention: Includes only the sequential convolution layer, with all attention mechanisms removed.

The ablation results are shown in Fig. 6. First, the model with all attention mechanisms removed (r.m. Parallel Attention) performs the worst across all evaluation metrics, indicating the baseline performance of the model without any attention enhancement. Second, when either the channel or spatial attention module is removed, the model shows inferior performance in terms of loss, but still outperforms the variant without any attention module in terms of accuracy and weighted F1-score (r.m. Spatial Attention and r.m. Channel Attention). This demonstrates that incorporating either attention mechanism individually can still contribute positively to model performance. Finally, the complete VulSCP model achieves the best results across all metrics, confirming that the parallel attention mechanism significantly enhances the model's performance, particularly in accuracy and F1-score. This indicates that the parallel attention layer can more effectively extract and leverage the features produced by the sequential convolution layer.

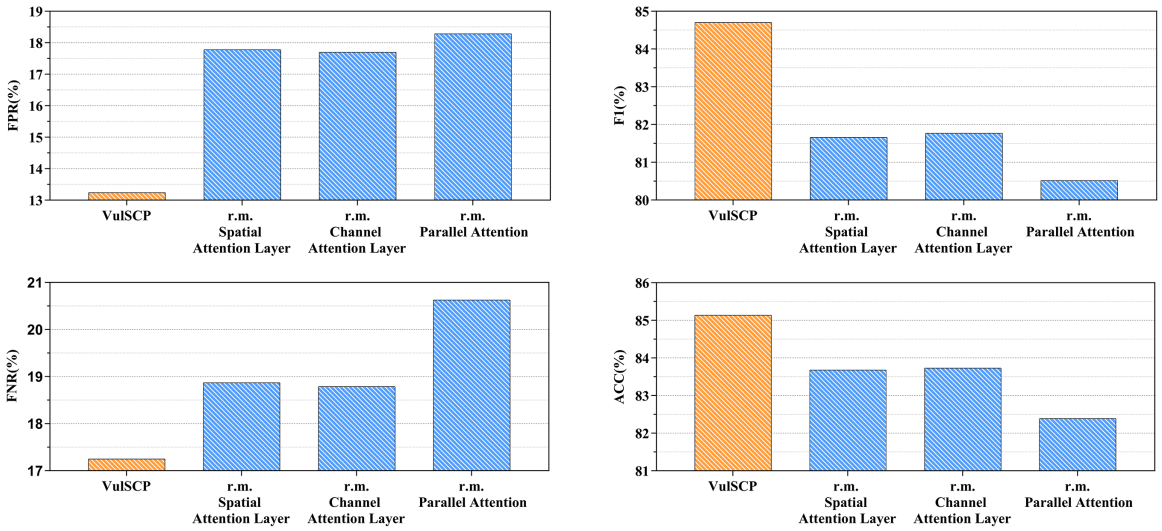


Figure 6: Results of the ablation experiments.

In conclusion, the experimental results suggest that the parallel attention mechanism, including both its channel and spatial attention components, contributes substantially to improving the performance of vulnerability detection.

5.3.2 Ablation on Centrality Weighting

To further validate the independent contribution of the centrality weighting strategy, we conduct an additional ablation study on the three centrality components used in VulSCP, namely Degree, Closeness, and Katz. In our model, each centrality measure is introduced as an independent weighting channel to modulate the importance of program nodes. To remove a specific centrality component, we set the centrality values of *all* nodes in the corresponding channel to 1, which yields uniform weights and thus disables node-importance differentiation by that centrality, while keeping the rest of the model unchanged. Accordingly, we construct the following variants for comparison:

- VulSCP: The full model with all three centrality weighting channels.
- r.m. Degree: Sets all node Degree-centrality values to 1 in the Degree channel.
- r.m. Closeness: Sets all node Closeness-centrality values to 1 in the Closeness channel.

- r.m. Katz: Sets all node Katz-centrality values to 1 in the Katz channel.
- r.m. All: Sets all three centrality channels to 1.

As shown in Table 4, removing any single centrality component leads to a performance drop, indicating that each component provides useful structural information for vulnerability detection. Compared with the complete VulSCP model, r.m. Degree reduces ACC from 85.14% to 83.68% and F1-score from 84.71% to 84.08%, while FPR and FNR increase to 17.96% and 14.73%, respectively. Similarly, r.m. Closeness lowers ACC and F1-score to 83.83% and 84.29%, and r.m. Katz lowers them to 84.18% and 84.52%.

Table 4: Ablation study on centrality weighting. All results are reported as mean ± standard deviation over five runs. The best results are **bolded** and the second-best are underlined.

Variants	ACC (%)	F1 Score (%)	FPR (%)	FNR (%)
VulSCP	$\mathbf{85.14_{\pm 0.32}}$	$\mathbf{84.71_{\pm 0.37}}$	$\mathbf{17.25_{\pm 0.58}}$	$\mathbf{13.24_{\pm 0.27}}$
r.m. Degree	$83.68_{\pm 0.41}$	$84.08_{\pm 0.45}$	$17.96_{\pm 0.63}$	$14.73_{\pm 0.35}$
r.m. Closeness	$83.83_{\pm 0.39}$	$84.29_{\pm 0.42}$	$\underline{17.54}_{\pm 0.61}$	$14.86_{\pm 0.38}$
r.m. Katz	$\underline{84.18}_{\pm 0.36}$	$\underline{84.52}_{\pm 0.39}$	$17.68_{\pm 0.55}$	$\underline{13.99}_{\pm 0.31}$
r.m. All	$82.48_{\pm 0.46}$	$82.72_{\pm 0.52}$	$19.87_{\pm 0.72}$	$15.12_{\pm 0.43}$

Among the three single-removal variants, r.m. Degree causes the largest drop in ACC and the highest FPR, suggesting that degree centrality contributes more to overall discrimination and false-positive suppression. r.m. Closeness yields the highest FNR (14.86%), indicating that closeness centrality is more helpful for reducing missed detections of vulnerable samples. Although the effect of removing Katz is relatively smaller, its performance is still consistently worse than that of the full model, which shows that Katz centrality also provides complementary structural cues.

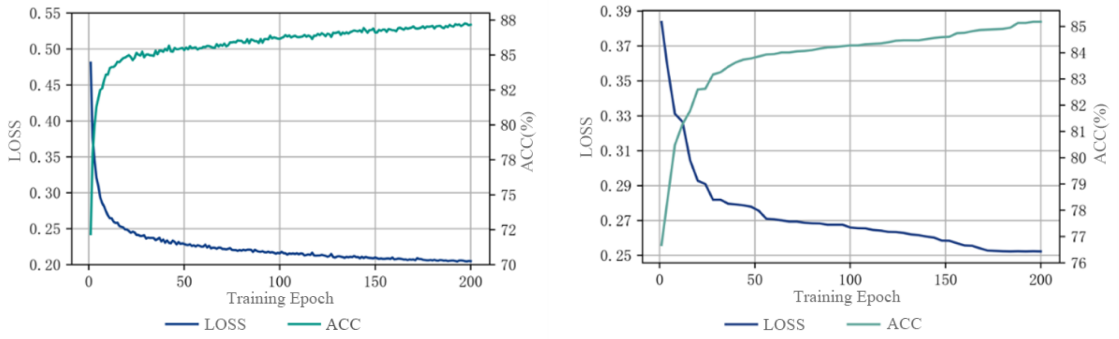
The largest degradation appears in r.m. All, where all centrality channels are removed. In this case, ACC drops to 82.48% and F1-score drops to 82.72%, which are 2.66 and 1.99 percentage points lower than those of VulSCP, respectively. At the same time, FPR rises from 17.25% to 19.87%, and FNR rises from 13.24% to 15.12%. These results demonstrate that the centrality weighting strategy is an effective component of VulSCP, and that combining multiple centrality measures helps the model better capture critical structural information for vulnerability detection.

5.4 Model Validation and Sensitivity Analysis

5.4.1 Convergence Analysis

The experimental procedure consists of two parts: (1) validating the convergence of accuracy and loss values with the window size set to 100, and (2) evaluating the impact of different window sizes on detection accuracy. To verify the effectiveness of the proposed method, this section validates the convergence of accuracy and loss using the partitioned dataset.

As shown in Fig. 7a, on the training dataset, both accuracy and loss values steadily converge as training epochs progress. Notably, they converge rapidly within the first 50 epochs, approaching optimal performance early in the training process. In contrast, Fig. 7b illustrates that on the validation dataset, convergence is also observed, but at a slower pace, with peak performance achieved around the 175th epoch. These experimental results demonstrate the effectiveness of the proposed VulSCP framework.

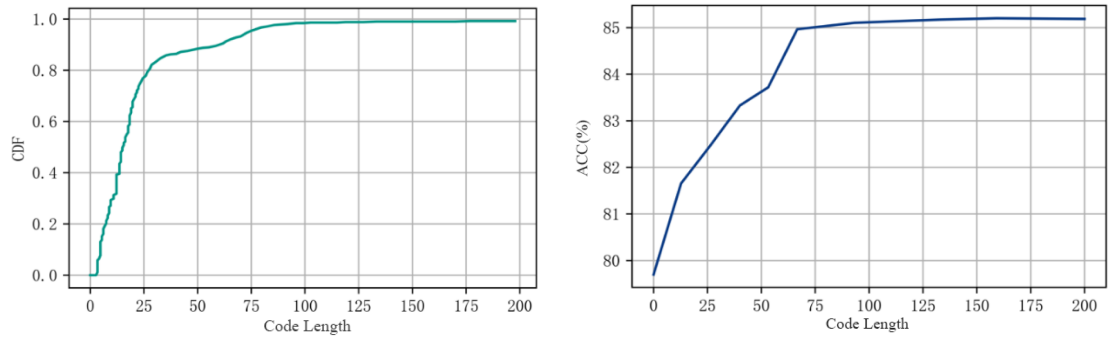


a Training. **b** Validation.

Figure 7: Convergence of loss and accuracy during training and validation.

5.4.2 Window Size Sensitivity Analysis

This section analyzes the effect of varying window sizes on the accuracy of vulnerability detection. To understand the distribution of function lengths in the combined dataset used in this study, we compute the Cumulative Distribution Function (CDF) of the code length across all samples. The CDF illustrates the probability that a variable takes a value less than or equal to a specified threshold, thus reflecting the cumulative distribution of the data. The result is shown in Fig. 8a; over 99% of function samples contain fewer than 200 lines of code. Based on this observation, we set 200 lines as the upper limit for the window size and evaluate the model's detection accuracy across different window lengths.



a CDF of sample lengths. **b** Performance for different thresholds.

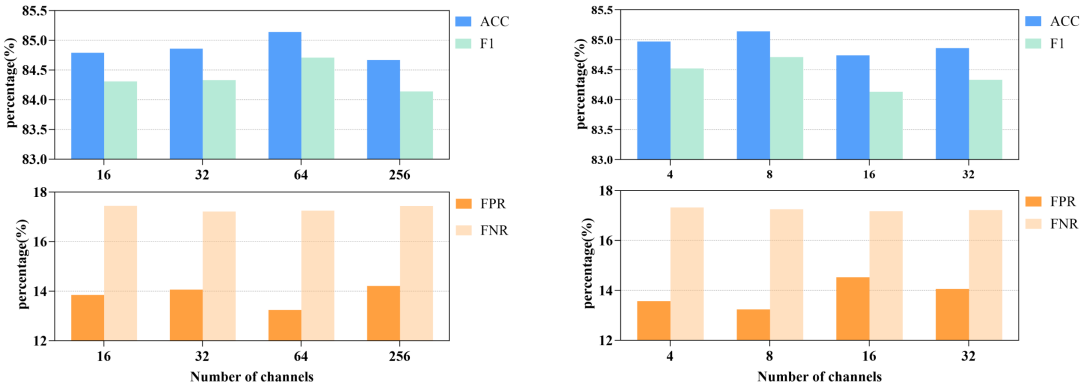
Figure 8: CDF of sample lengths and detection performance under different thresholds.

The evaluation results in Fig. 8b indicate a positive correlation between window size and detection accuracy. Specifically, accuracy improves as the window size increases, and the improvement becomes marginal beyond 100 lines, where the accuracy approaches its optimal value and the growth rate significantly slows down.

5.4.3 Hyperparameter Sensitivity Analysis

This section presents hyperparameter experiments targeting the parallel attention mechanism in VulSCP.

(1) Number of input channels in the attention layer. The number of input channels in the parallel attention layer reflects the depth of high-level code feature refinement performed by the model. We evaluated four different channel sizes: 16, 32, 64, and 256. As shown in Fig. 9a and Table 5, the model's loss exhibits an irregular trend with increasing channel numbers. Specifically, when the number of channels is 64, the model achieves optimal performance across most evaluation metrics, including the lowest false negative rate of 13.24%, the highest F1 score of 84.71%, and the highest accuracy of 85.14%. This suggests that increasing the channel width enhances the model's capacity for feature representation, thereby improving its detection performance. However, when the number of channels reaches 256, performance declines across all metrics, indicating potential overfitting.



a Attention input channels. **b** Attention groups.

Figure 9: Performance across different attention input channel sizes and group sizes.

Table 5: Parametric experiment on attention channels. All results are reported as mean ± standard deviation over five runs. The best results are **bolded** and the second-best are underlined.

Channels	Accuracy (%)	F1 Score (%)	FPR (%)	FNR (%)
16	$84.79_{\pm 0.35}$	$84.31_{\pm 0.41}$	$\underline{17.45}_{\pm 0.62}$	$\underline{13.85}_{\pm 0.33}$
32	$\underline{84.86}_{\pm 0.34}$	$\underline{84.33}_{\pm 0.39}$	$\mathbf{17.22_{\pm 0.59}}$	$14.06_{\pm 0.34}$
64	$\mathbf{85.14_{\pm 0.32}}$	$\mathbf{84.71_{\pm 0.37}}$	$17.25_{\pm 0.58}$	$\mathbf{13.24_{\pm 0.27}}$
256	$84.67_{\pm 0.38}$	$84.14_{\pm 0.43}$	$17.44_{\pm 0.65}$	$14.21_{\pm 0.36}$

(2) Number of parallel attention groups. Grouping channels in the parallel attention layer can reduce computational overhead, but excessive grouping may hinder the learning of global features. We tested four group sizes: 4, 8, 16, and 32. As shown in Fig. 9b and Table 6, the model performs best when the group number is set to 8, achieving the lowest false negative rate of 13.24%, the highest weighted F1 score of 84.71%, and the highest accuracy of 85.14%. This suggests that an appropriate grouping factor can balance model complexity and generalization capability, preserving sufficient feature extraction power while avoiding overfitting. When the group number increases to 16 and 32, the accuracy remains relatively high, but the performance on other metrics shows no significant improvement.

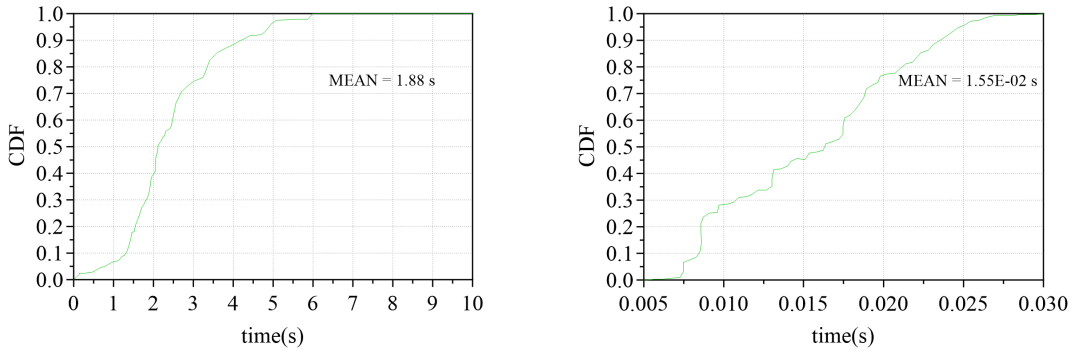
Table 6: Parametric experiment on attention groups. All results are reported as mean ± standard deviation over five runs. The best results are **bolded** and the second-best are underlined.

Group Quantity	Accuracy (%)	F1 Score (%)	FPR (%)	FNR (%)
4	$84.97_{\pm 0.33}$	$84.52_{\pm 0.38}$	$17.32_{\pm 0.61}$	$13.57_{\pm 0.31}$
8	$\mathbf{85.14_{\pm 0.32}}$	$\mathbf{84.71_{\pm 0.37}}$	$17.25_{\pm 0.58}$	$\mathbf{13.24_{\pm 0.27}}$
16	$84.74_{\pm 0.36}$	$84.13_{\pm 0.42}$	$\mathbf{17.17_{\pm 0.57}}$	$14.53_{\pm 0.38}$
32	$84.86_{\pm 0.35}$	$84.33_{\pm 0.40}$	$17.22_{\pm 0.63}$	$14.06_{\pm 0.35}$

5.5 Efficiency and Computational Complexity Analysis

5.5.1 Inference Time Evaluation

To further evaluate the efficiency of the proposed approach, we analyze the runtime from sample input to vulnerability detection output. The execution efficiency is assessed by plotting the cumulative distribution function (CDF) curve of runtime. During the data preprocessing phase, as illustrated in Fig. 10a, the extraction of semantically weighted graphs accounts for a more significant portion of the total runtime. Although the tail of the distribution extends to around 5 s, the preprocessing time is considered acceptable for the overall system overhead. In the classification phase, as shown in Fig. 10b, the total processing time for all test samples ranges between 0.005 and 0.030 s. Approximately 80% of test samples are processed within 0.020 s, while about 50% complete classification in under 0.015 s, achieving millisecond-level processing time per sample.



a Overhead of Data Pre-processing (s). **b** Overhead of Classification (s).

Figure 10: Runtime analysis results.

In summary, the proposed method maintains an average end-to-end runtime of 1.89 s, demonstrating the high efficiency of the VulSCP model.

5.5.2 Computational Complexity Comparison

To further quantify the computational complexity and resource overhead in practical deployment, we provide a cost profile of VulSCP and compare it with AugSliceVul, the strongest baseline in our main and cross-dataset experiments. Specifically, we report model size (Total Params and Weight Size), arithmetic cost (FLOPs per sample), and runtime GPU footprint (peak inference/training GPU memory) under two batch sizes. The results are summarized in Table 7.

Table 7: Computational complexity comparison between VulSCP and AugSliceVul under different batch sizes.

Method	Batch	Static Model Complexity Metrics			Runtime GPU Memory Footprint	
		Total Params	FLOPs (G/sample)	Weight Size (MB)	Peak Inference (MB)	Peak Training (MB)
VulSCP	16	732,226	0.1272	2.79	17.47	45.79
	32				21.66	52.54
AugSliceVul	16	124,647,170	263.1112	475.52	764.93	5440.57
	32				1045.11	10,358.88

From a static complexity perspective, VulSCP is lightweight (0.73M parameters, 2.79 MB weights) and requires 0.1272 G FLOPs per sample, whereas AugSliceVul is substantially larger (124.65M parameters, 475.52 MB weights) and more compute-intensive (263.11 G FLOPs per sample). This corresponds to roughly ~170× fewer parameters/weights and ~2000× fewer FLOPs, indicating that the improvements reported in Sections 5.1 and 5.2 are achieved with an efficient architecture rather than by scaling model size.

The GPU memory footprint exhibits a similarly favorable pattern. Across both batch sizes, VulSCP stays below 22 MB peak inference memory and below 53 MB peak training memory, while AugSliceVul requires 0.76–1.05 GB for inference and 5.44–10.36 GB for training. Moreover, when doubling the batch size from 16 to 32, AugSliceVul shows a much steeper training-memory increase (90.4%) than VulSCP (14.7%), suggesting that VulSCP provides more stable batch scalability under limited GPU budgets.

Overall, VulSCP demonstrates a strong accuracy–efficiency trade-off: compared with the best-performing baseline, it achieves markedly lower computational and memory costs while delivering better detection performance.

5.6 Limitations and Practical Considerations

Although VulSCP achieves strong overall performance, several limitations should be noted. First, despite its relatively low false positive rate, the false negative rate in some settings indicates that certain vulnerability patterns may still not be fully captured by the current model. Second, although we further evaluate cross-dataset generalization, the present experimental setting is still based mainly on random data splitting and therefore cannot fully reflect real industrial deployment conditions. Third, the current study focuses on function-level vulnerability detection, which may be insufficient for vulnerabilities involving cross-function, cross-file, or cross-module semantic dependencies. These issues will be further explored in future work.

5.7 Case Study

VulSCP is a vulnerability detection method built on graph-derived semantic representation, which allows visualization techniques to be used to explain detection results. In this section, we take buffer overflow vulnerabilities as an example and apply the Gradient-weighted Class Activation Mapping++ (Grad-CAM++) method to visualize the model's prediction outcomes.

Fig. 11 illustrates the visualization results for buffer overflow vulnerability detection. Code lines 8 and 18 are highlighted in the deepest red, followed by lines 7 and 11 in a slightly lighter shade, indicating that these lines are likely to contain vulnerabilities.

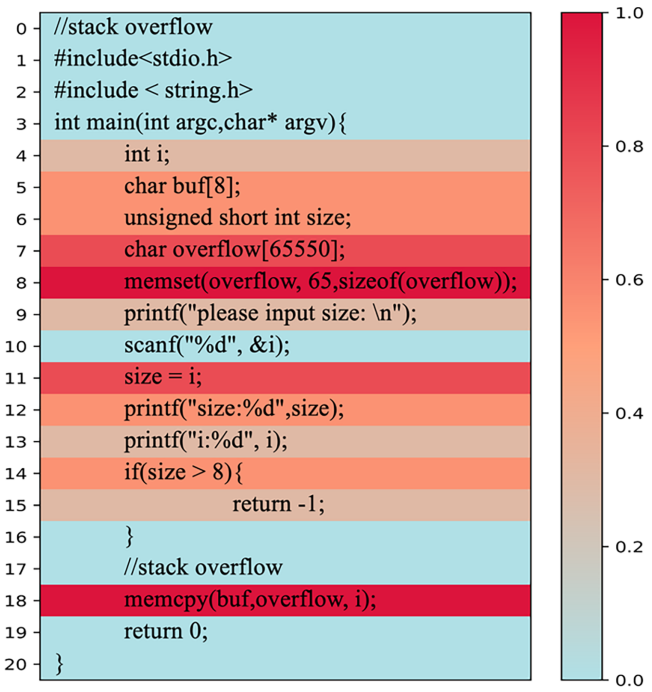


Figure 11: Risk-level visualization of code vulnerability detection using Grad-CAM++.

In this example, line 7 allocates a character array named overflow with a size of 65,550 bytes on the stack. Given the limited stack space, this may lead to a stack overflow. At line 10, the program reads an integer from user input via scanf, assigns it to i, and subsequently to size. If the user inputs an excessively large integer (exceeding the capacity of an unsigned short int), an integer overflow may occur. Finally, the memcpy function copies data from the overflow array to the buf array without boundary checks, potentially causing a buffer overflow.

In summary, the visualization results generated by VulSCP align well with the expected outcomes of manual code analysis, thereby supporting the effectiveness of the proposed method.

6 Conclusion

This paper presents VulSCP, a code vulnerability detection framework that combines semantically weighted graph representation, sequential convolution, and parallel attention. By preserving control-flow and data-flow semantics through PDG-based representation and further enhancing feature learning with centrality weighting and parallel attention, the proposed framework improves the modeling of vulnerability-relevant patterns and global semantic dependencies.

Experimental results show that VulSCP performs favorably relative to recent and representative vulnerability detection baselines, while maintaining a good balance between semantic modeling capability and computational efficiency. These findings suggest that the proposed framework is a promising approach for efficient vulnerability detection in large-scale code settings.

Despite these results, there remains room for improvement in reducing the false negative rate. In addition, the current study focuses on function-level vulnerability detection, and further validation under broader code contexts and more realistic deployment scenarios is still needed. These issues will be explored in future work.

Acknowledgement: Not applicable.

Funding Statement: This research was funded by the Ministry of Public Security of the People's Republic of China, grant number 2024ZB02 (X.Z.).

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Zhe Wang, Yu Yan and Junqi Tong; methodology, Zhe Wang, Yu Yan and Junqi Tong; software, Zhe Wang and Yu Yan; validation, Zhe Wang, Yu Yan, Junqi Tong and Yijun Lin; formal analysis, Zhe Wang and Yu Yan; investigation, Zhe Wang, Yu Yan, Junqi Tong and Yijun Lin; resources, Dechun Yin and Xiaoliang Zhao; data curation, Zhe Wang and Yu Yan; writing—original draft preparation, Zhe Wang, Yu Yan, Junqi Tong and Yijun Lin; writing—review and editing, all authors; visualization, Zhe Wang, Yu Yan, Junqi Tong and Yijun Lin; supervision, Dechun Yin and Xiaoliang Zhao; project administration, Xiaoliang Zhao; funding acquisition, Xiaoliang Zhao. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Ethics Approval: Not applicable.

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

References

1. National Institute of Standards and Technology (NIST). National vulnerability database [Internet]. 2025 [cited 2026 Mar 21]. Available from: https://nvd.nist.gov/.
2. SecPod. The cybersecurity landscape of 2024: key insights from the annual vulnerability report [Internet]. 2024 [cited 2026 Mar 21]. Available from: https://www.secpod.com/blog/the-cybersecurity-landscape-of-2024-key-insights-from-the-annual-vulnerability-report/.
3. Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire OS distributions. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P). Piscataway, NJ, USA: IEEE; 2012. p. 48–62.
4. Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discovery. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P). Piscataway, NJ, USA: IEEE; 2017. p. 595–614.
5. Li J, Ernst MD. CBCD: cloned buggy code detector. In: Proceedings of the 34th International Conference on Software Engineering (ICSE). Piscataway, NJ, USA: IEEE; 2012. p. 310–20.
6. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, et al. VulDeePecker: a deep learning-based system for vulnerability detection. arXiv:1801.01681. 2018.
7. Lin G, Zhang J, Luo W, Pan L, Xiang Y. POSTER: vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). New York, NY, USA: ACM; 2017. p. 2539–41.
8. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans Depend Secure Comput. 2021;19(4):2244–58.
9. Su X, Zheng W, Jiang Y, Wei H, Wan J, Wei Z. Research and progress on learning-based source code vulnerability detection. Chin J Comput. 2024;47:337–74.
10. Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Red Hook, NY, USA: Curran Associates Inc.; 2019. p. 10197–207.
11. Wang M, Tao C, Guo H. LCVD: loop-oriented code vulnerability detection via graph neural network. J Syst Softw. 2023;202:111706. doi:10.2139/ssrn.4328066.
12. Shao M, Ding Y, Cao J, Li Y. GraphFVD: property graph-based fine-grained vulnerability detection. Comput Secur. 2025;151:104350. doi:10.1016/j.cose.2025.104350.

13. Zhang QL, Yang YB. SA-Net: shuffle attention for deep convolutional neural networks. In: ICASSP 2021—2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). Piscataway, NJ, USA: IEEE; 2021. p. 2235–9.
14. Smaili A, Zhang Y, Mekkaoui DE, Midoun MA, Talhaoui MZ, Hamidaoui M, et al. A transformer-based framework for software vulnerability detection using attention-driven convolutional neural networks. Eng Appl Artif Intell. 2025;160(9):111859. doi:10.1016/j.engappai.2025.111859.
15. Hussain S, Nadeem M, Baber J, Hamdi M, Rajab A, Al Reshan MS, et al. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. Sci Rep. 2024;14(1):7406. doi:10.1038/s41598-024-56871-z.
16. Du X, Wen M, Zhu J, Xie Z, Ji B, Liu H, et al. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. arXiv:2406.03718. 2024.
17. Yang AZ, Tian H, Ye H, Martins R, Goues CL. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. arXiv:2406.05892. 2024.
18. Sejfia A, Das S, Shafiq S, Medvidović N. Toward improved deep learning-based vulnerability detection. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. New York, NY, USA: ACM; 2024. p. 1–12.
19. Wheeler DA. Flawfinder [Internet]. 2001 [cited 2026 Mar 21]. Available from: https://dwheeler.com/flawfinder/.
20. Marjamäki D. Cppcheck [Internet]. 2007 [cited 2026 Mar 21]. Available from: http://cppcheck.sourceforge.net/.
21. Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J. Using FindBugs on production software [Internet]. 2008 [cited 2026 Mar 21]. Available from: https://findbugs.sourceforge.net.
22. Clang Static Analyzer. Clang static analyzer [Internet]. 2020 [cited 2026 Mar 21]. Available from: https://clang-analyzer.llvm.org/.
23. Micro Focus. Fortify static code analyzer [Internet]. 2019 [cited 2026 Mar 21]. Available from: https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview.
24. Checkmarx. Checkmarx [Internet]. 2021 [cited 2026 Mar 21]. Available from: https://www.checkmarx.com.
25. RATS. Rough auditing tool for security (RATS) [Internet]. 2021 [cited 2026 Mar 21]. Available from: https://code.google.com/archive/p/rough-auditing-tool-for-security/.
26. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, et al. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). Piscataway, NJ, USA: IEEE; 2018. p. 757–62.
27. Duan X, Wu J, Ji S, Rui Z, Luo T, Yang M, et al. Focus your attention to shoot fine-grained vulnerabilities. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19); 2019 Aug 10–16; Macao, China. p. 4665–71.
28. Feng H, Fu X, Sun H, Wang H, Zhang Y. Efficient vulnerability detection based on abstract syntax tree and deep learning. In: IEEE INFOCOM 2020—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway, NJ, USA: IEEE; 2020. p. 722–7.
29. Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H. VulDeeLocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans Depend Secure Comput. 2021;19(4):2821–37. doi:10.1109/tdsc.2021.3076142.
30. Zhou X, Zhang T, Lo D. Large language model for vulnerability detection: emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. New York, NY, USA: ACM; 2024. p. 47–51.
31. Fu M, Tantithamthavorn CK, Nguyen V, Le T. ChatGPT for vulnerability detection, classification, and repair: how far are we?. In: 2023 30th Asia-Pacific Software Engineering Conference (APSEC). Piscataway, NJ, USA: IEEE; 2023. p. 632–6.
32. Yin X. Pros and cons! Evaluating ChatGPT on software vulnerability. arXiv:2404.03994. 2024.
33. Khare A, Dutta S, Li Z, Solko-Breslin A, Alur R, Naik M. Understanding the effectiveness of large language models in detecting security vulnerabilities. In: 2025 IEEE Conference on Software Testing, Verification and Validation (ICST). Piscataway, NJ, USA: IEEE; 2025. p. 103–14.

34. Zhang C, Liu H, Zeng J, Yang K, Li Y, Li H. Prompt-enhanced software vulnerability detection using ChatGPT. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. New York, NY, USA: ACM; 2024. p. 276–7.
35. Ni C, Shen L, Xu X, Yin X, Wang S. Learning-based models for vulnerability detection: an extensive study. arXiv:2408.07526. 2024.
36. Zhou X, Tran DM, Le-Cong T, Zhang T, Irsan IC, Sumarlin J, et al. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. arXiv:2407.16235. 2024.
37. Yang X, Wang S, Li Y, Wang S. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Piscataway, NJ, USA: IEEE; 2023. p. 2287–98.
38. Ding Y, Fu Y, Ibrahim O, Sitawarin C, Chen X, Alomair B, et al. Vulnerability detection with code language models: how far are we? arXiv:2403.18624. 2024.
39. Wen XC, Wang X, Gao C, Wang S, Liu Y, Gu Z. When less is enough: positive and unlabeled learning model for vulnerability detection. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway, NJ, USA: IEEE; 2023. p. 345–57.
40. Liu Z, Tang Z, Zhang J, Xia X, Yang X. Pre-training by predicting program dependencies for vulnerability analysis tasks. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Piscataway, NJ, USA: IEEE; 2024. p. 1–13.
41. Peng T, Chen S, Zhu F, Tang J, Liu J, Hu X. PTLVD: program slicing and transformer-based line-level vulnerability detection system. In: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM). Piscataway, NJ, USA: IEEE; 2023. p. 162–73.
42. Zhang J, Liu Z, Hu X, Xia X, Li S. Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans Softw Eng. 2023;49(8):4196–212. doi:10.1109/tse.2023.3286586.
43. Jiang Z, Sun W, Gu X, Wu J, Wen T, Hu H, et al. DFEPT: data flow embedding for enhancing pre-trained model based vulnerability detection. In: Proceedings of the 15th Asia-Pacific Symposium on Internetware. New York, NY, USA: ACM; 2024. p. 95–104.
44. Tang W, Tang M, Ban M, Zhao Z, Feng M. CSGVD: a deep learning approach combining sequence and graph embedding for source code vulnerability detection. J Syst Softw. 2023;199:111623.
45. Ziems N, Wu S. Security vulnerability detection using deep learning natural language processing. In: IEEE INFOCOM 2021—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway, NJ, USA: IEEE; 2021. p. 1–6.
46. National Institute of Standards and Technology (NIST). Software assurance reference dataset [Internet]. 2025 [cited 2026 Mar 21]. Available from: https://samate.nist.gov/SARD/.
47. Fan J, Li Y, Wang S, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. New York, NY, USA: ACM; 2020. p. 508–12. doi:10.1145/3379597.3387501.
48. Lin G, Xiao W, Zhang J, Xiang Y. Deep learning-based vulnerable function detection: a benchmark. In: Information and communications security. Cham, Switzerland: Springer International Publishing; 2020. p. 219–32. doi:10.1007/978-3-030-41579-2_13.
49. Zou Z, Jiang T, Wang Y, Xue T, Zhang N, Luan J. Code vulnerability detection based on augmented program dependency graph and optimized CodeBERT. Sci Rep. 2025;15(1):39301. doi:10.1038/s41598-025-23029-4.