BAVul: Code Vulnerability Detection based on Bug Attention

Chunyu Yang[1], Bo Zhao[1*]
[1] School of Cyber Science and Engineering, Wuhan University, Wuhan, China
yangchunyu@whu.edu.cn, zhaobo@whu.edu.cn

***Abstract*—Machine learning is widely applied in software engineering tasks, particularly code vulnerability detection, owing to its ability to automatically identify patterns in data. However, existing machine-learning-based vulnerability detection methods exhibit several limitations. Reliance on a single feature type fails to capture comprehensive code semantics, leading to elevated false positive and false negative rates. Excessive dependence on fully automated feature learning overlooks potential inter-feature relationships, thereby impeding the model's capacity to recognize specific vulnerability patterns. Furthermore, the use of excessively high-dimensional features incurs substantial computational overhead during large-scale analysis, rendering real-time detection impractical. To address these shortcomings, we propose BAVul, a vulnerability detection system based on a bug attention mechanism. BAVul embeds source code using CodeBert language model to preserve vulnerability-relevant semantic information in the sequence. It then reconstructs the Control Flow Graph (CFG), Abstract Syntax Tree (AST), and Program Dependence Graph (PDG), and generates embeddings for these graphs via Graph Attention Network (GAT) to preserve structural information. During feature fusion, a bug attention mechanism is introduced to enable weighted integration of graph embeddings, thereby compensating for the limited modeling of vulnerability patterns in conventional graph-based approaches. In the detection phase, a refinement-based dimensionality reduction technique is combined with LightGBM to enhance both efficiency and accuracy. Compared with conventional machine-learning methods, BAVul achieves substantial improvements in F1-score and computational performance.**

***Index Terms*—Machine Learning, Vulnerability, Cross-Attention, Token, Graph**

I. Introduction

Software vulnerabilities remain one of the primary sources of cybersecurity risks today. Exploitation of these vulnerabilities frequently results in severe consequences, including massive data breaches, full system compromise, ransomware attacks, and even disruption of critical infrastructure. Vulnerability detection system plays a vital role in software security [1], which can prevent a series of security incidents [2]. Consequently, interest in more accurate and efficient automated software vulnerability detection methods has increased [3].

However, existing machine learning-based code vulnerability detection methods still face limitations in feature processing [4]–[8]. First, single-feature enhancement approaches suffer from semantic limitations and fail to comprehensively capture diverse vulnerability patterns. Second, multi-feature fusion methods often ignore dependencies among internal features, limiting the model's ability to incorporate contextual influences during learning. As a result, these methods overlook potential semantic correlations across different feature representations (such as those between source code and program dependence graphs) and rely solely on basic feature concatenation. This leads to feature redundancy, difficulty in capturing vulnerability-specific patterns, and increased false positive and false negative rates. Furthermore, the high-dimensional feature vectors introduced by existing approaches frequently cause overfitting during training and fail to meet real-time detection requirements.

To address the aforementioned limitations, we propose BAVul, a code vulnerability detection system based on a bug attention mechanism. BAVul integrates sequence-level and graph-level features to deliver richer semantic information than existing approaches. In BAVul, CodeBERT first tokenizes and embeds the source code to produce token-level feature vectors, the GAT is then applied to embed the three code structural representations: CFG, AST, and PDG. The token embedding then serve as queries in the bug attention mechanism, identifying important structure information in the graph embeddings and assigning token related weights to achieve semantic enhancement of vulnerability patterns. Finally, a distance-based feature dimensionality reduction method processes the fused features, which are then fed into a LightGBM model for efficient and accurate vulnerability detection. The main contributions of this work are as follows:

1) Multi-type Feature Recovery. To address the issue of insufficient semantic information caused by single-feature sources, we extract both textual and graph-level features from the code. Considering the characteristics of text and graph representations, we use a token embedding module and a graph embedding module to separately extract semantic information from the two types of features.

2)Bug Attention-Based Semantic Enhancement. To overcome the limitation that features cannot leverage information from other features during the fusion process, we propose bug attention, a mechanism that uses token embeddings as queries and graph embeddings as keys and values in a cross-attention framework. This allows the tokens to semantically reinforce the graph, capturing richer vulnerability pattern features from

This research is funded by the National Natural Science Foundation of China (Grant No. U1936122, 62572356) and the Primary Research & Development Plan of Hubei Province (Grant No. 2020BAA003). * Indicates corresponding author.

the graph.

3) To mitigate overfitting and excessive computational cost caused by high-dimensional features, we refine the fused features using six distance metrics and then perform vulnerability detection with LightGBM.

II. BackGround

A. Dynamic Analysis

Dynamic analysis methods typically involve inserting instrumentation code into the target program and executing it. By integrating with existing fuzzing frameworks such as AFL [9], these methods generate test cases for the target code. Feedback from the instrumentation is used to iteratively refine the test cases, thereby enhancing testing efficiency. Ultimately, test cases that cause crashes are audited to identify potential security vulnerabilities. However, these approaches heavily depend on the fuzzing framework's exploration of code paths and the program's internal execution, rendering them extremely time-consuming. Discovering vulnerabilities via dynamic analysis often requires several days, making it inadequate for the escalating demands of code analysis [10], [11].

B. Static Analysis

According to their underlying principles, static analysis techniques can be categorized into taint analysis, symbolic execution, and BCSD-based methods. Taint analysis first identifies sensitive program points (sinks) and data input points (sources) in the code. It then tracks taint propagation from sources to sinks, checking for unsafe paths that may allow untrusted input to reach sensitive operations [12], [13]; potential vulnerabilities are subsequently confirmed through manual auditing. Symbolic execution uses constraint solvers to explore different possible values at sensitive program points, thereby determining the feasibility of vulnerability-triggering conditions. BCSD-based methods are primarily divided into distance-based approaches and machine learning-based approaches [14], [15]. Distance-based methods identify similar features between vulnerable code fragments, compute similarity metrics, and apply thresholds to detect potentially vulnerable code. Machine learning-based methods focus on extracting semantic features—such as textual or structural representations of code—and leverage model learning capabilities to uncover latent vulnerability patterns for detection.

Taint analysis and symbolic execution require exhaustive traversal and processing of a large number of code paths, incurring substantial computational time across diverse program states. Moreover, they suffer from issues such as over-tainting and path explosion [16], [17], rendering them insufficient for large-scale analysis demands. In contrast, machine learning-based methods can automatically extract semantic features and have thus become the predominant approach for code vulnerability detection.

III. Proposed System

The overview of BAVul is showed in Figure. 1. BAVul comprises three primary stages: token and graph embedding, bug attention and vulnerability detection. In the first phase, we introduce two levels of code features. At the token level, we perform tokenization on the source code to obtain code tokens and extract semantic features at the code sequence level using existing pretrained code models. At the graph level, we employ a graph embedding network to capture semantic features from three kinds of graph structures.

In the second phase, we propose a Bug Attention module. To further emphasize vulnerability-relevant patterns in the graph semantics, we use the token embeddings as queries, the graph embeddings as keys and values, and compute cross-attention from tokens to the graph. This yields an enhanced graph embedding.

In the final phase, to mitigate the overfitting and computational cost during detection caused by excessively high feature dimensionality, we combine six distance metrics to refine the features, yielding lower-dimensional representations with enhanced semantics. We then train a LightGBM classifier to identify vulnerability sample.

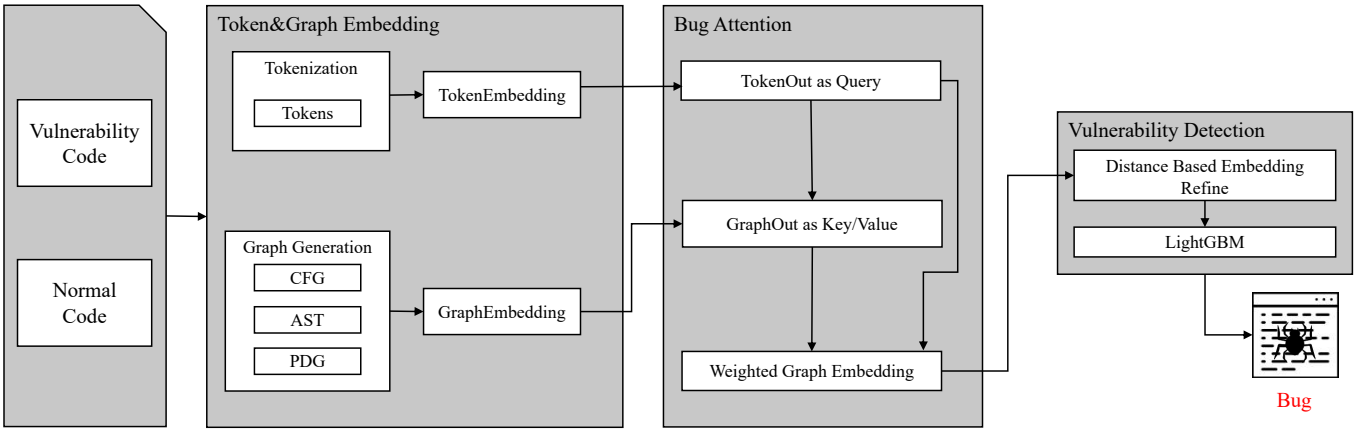


Fig. 1: BAVul Overview.

A. Embedding

1) Code Embedding: As shown in Figure. 2, the code embedding process consists of two main stages: tokenization and vectorization. To obtain code sequence embeddings, we first perform tokenization on the source code. Common tokenization approaches include word-level, character-level, and subword tokenization. Word-level tokenization suffers from out-of-vocabulary (OOV) issues, while character-level tokenization produces excessively long sequences. Consequently, neither is suitable for vulnerability detection in code. In this work, we adopt subword tokenization for processing source code. WordPiece and Byte Pair Encoding (BPE) are two widely used subword tokenizers. WordPiece has the drawback of generating [UNK] tokens, which may cause loss of vulnerability-related information. Therefore, we employ BPE as the tokenization tool. BPE starts from a fixed set of 256 UTF-8 bytes, ensuring complete coverage of any input text.

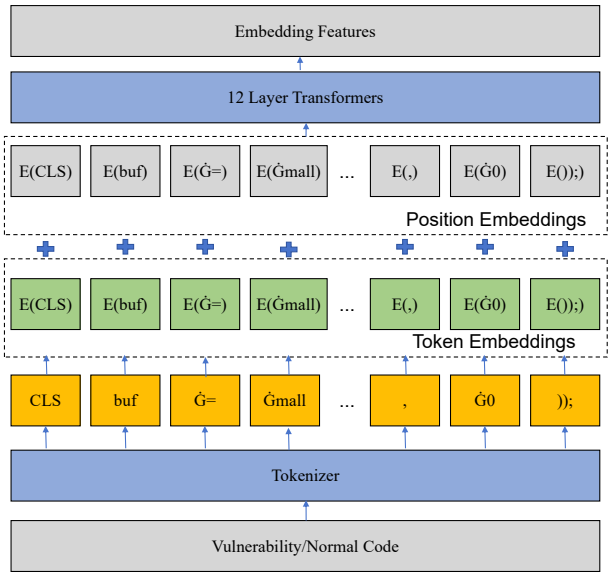


Fig. 2: Code Embedding Process.

To convert token sequences into numerical vectors, we adopt the approach used in CodeBERT [18], which is built on the RoBERTa [19] architecture. First, each token is mapped to its corresponding ID in the vocabulary and then transformed into a dense vector through a learnable token embedding matrix, these token embeddings are then combined with position embeddings. The resulting representations are fed into transformer layers, where self-attention mechanisms enable each token to attend to all others, thereby capturing rich contextual relationships, as illustrated in Figure. 2. Finally, the model produces the embeddings from the last transformer layer, with each token encoded as a 768-dimensional vector that integrates both its local semantics and its global context within the code snippet.

2) Graph Embedding: To generate robust graph embeddings for vulnerability detection, we employ Graph Attention Networks (GAT) [20] to encode the multi-view structural representations of the code, specifically its CFG, AST, and PDG. For each graph, the initial node features are obtained from the CodeBERT embeddings of the tokens corresponding to each node.

The computation process is illustrated in Figure. 3. For each node N_i in the graph, the attention score between N_i and its neighbor N_j is computed using the following formula.

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{a}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_j}\exp\left(\text{LeakyReLU}\left(\vec{a}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \tag{1}$$

$\mathbf{W}$ is the weight matrix applied to the node features, $\mathbf{h}$ denotes the embedding vector of each node, $\vec{a}^\top$ is a learnable attention vector, and LeakyReLU serves as the activation function.

Finally, the embedding vector $\vec{h}'_i$ for node N_i is computed using the following formula.

$$\vec{h}'_i = \frac{1}{K}\sum_{k=1}^{K}\sigma\left(\sum_{j\in N_i}\alpha_{ij}^k\mathbf{W}^k\vec{h}_j\right) \tag{2}$$

Here, K denotes the number of attention heads, as illustrated by the three differently colored edge types in Figure. 3. To reduce the computational overhead in subsequent feature processing, we compute the node embedding $\vec{h}'_i$ by averaging the outputs across all attention heads rather than concatenating them.

This multi-graph GAT approach effectively leverages complementary information across control flow (CFG), syntactic hierarchy (AST), and data/control dependencies (PDG), enabling more discriminative feature learning for identifying vulnerable code patterns.

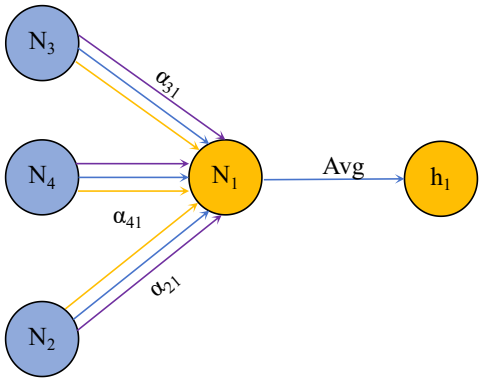


Fig. 3: Graph Attention Networks.

B. Bug Attention

To further enhance the semantic quality of the graph embedding, we propose a cross-attention-based feature fusion mechanism that integrates the previously obtained token embeddings with the graph embeddings.

In Section III-A, CodeBERT is employed to obtain the token-level embeddings of the code sequence, denoted as

$\mathbf{T} \in \mathbb{R}^{L \times D}$, where L is the sequence length and D is the embedding dimension (768 in this work). Subsequently, GAT is applied to three graph representations of the code (CFG, AST, and PDG) to produce the graph node embeddings $\mathbf{G} \in \mathbb{R}^{N \times D}$, where N denotes the number of nodes. We then employ a cross-attention mechanism to construct the bug attention module, which processes and fuses the two types of features. As illustrated in Figure. 4, we use the token embeddings as queries(Q), and the graph embeddings as both keys(K) and values(V) in a cross-attention mechanism. This computes an attention weight matrix A that reflects the importance of each node embedding to every token. The weight matrix A is then applied to the values to produce semantically enhanced graph features. Finally, a residual connection fuses these enhanced graph features with the original token embeddings.

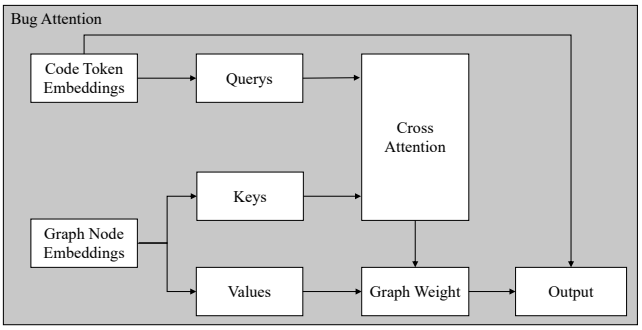


Fig. 4: The Process of Bug Attention.

Equation 3 describes the computation of A, where $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{D \times D_k}$ are learnable projection matrices. The scaling factor $\sqrt{D_k}$ in the denominator stabilizes the gradients and improves training stability.

$$\mathbf{A} = softmax\left(\frac{\mathbf{Q}\mathbf{W}_Q(\mathbf{K}\mathbf{W}_K)^\top}{\sqrt{D_k}}\right), \tag{3}$$

$\mathbf{H}$ is the output of the cross-attention layer, computed as the weighted sum of the value vectors (V) and $\mathbf{W}_V \in \mathbb{R}^{D \times D_v}$ using the attention weights (A). Its purpose is to produce semantically enriched representations for the graph nodes by dynamically aggregating relevant information from the code sequence, thereby injecting token information and importance weighting into the original graph structure.

$$\mathbf{H} = \mathbf{A}(\mathbf{V}\mathbf{W}_V), \tag{4}$$

The semantically enhanced graph embedding $\mathbf{H}'$ is finally computed via a residual connection. This residual connection preserves the original code semantics while enabling the model to incorporate structural information from the graph. By directly adding the cross-attention output to the initial code embeddings, the resulting representation effectively retains both the linguistic and structural features of the code.

$$\mathbf{H}' = \mathbf{H} + \mathbf{Q}. \tag{5}$$

This fusion aligns sequential context with graph nodes, emphasizing structurally relevant elements informed by the code's natural order and semantics. By injecting sequence-derived attention weights into CFG (control transitions), AST (syntactic hierarchy), and PDG (data/control dependencies), the approach yields more discriminative embeddings that better capture vulnerability patterns, such as context-dependent unsafe operations or anomalous data flows, ultimately improving detection performance in downstream classification tasks.

C. Vulnerability Detection

1) Distance Based Embedding Refine: To enhance the real-time detection capability of the model, we propose processing fused embeddings using distance metrics. Specifically, we first employ Jaccard similarity and Dice similarity to quantify the overlap in identical elements between the embeddings of two samples. To further capture sequential similarities, we introduce Levenshtein ratio and supplement it with Jaro similarity and Jaro-Winkler similarity to reinforce the semantic alignment in token order.

2) LightGBM Based Vulnerability Detection: LightGBM [21], a highly efficient gradient boosting framework developed by Microsoft, has been increasingly applied in vulnerability detection due to its superior speed, scalability, and strong performance on high-dimensional, imbalanced datasets commonly encountered in source code analysis. By leveraging histogram-based splitting, leaf-wise tree growth, and native support for categorical features, LightGBM effectively models complex vulnerability patterns extracted from features such as AST, CFG, PDG, or static code metrics, achieving competitive accuracy with significantly reduced training time and memory usage compared to traditional methods like XGBoost. This makes it particularly suitable for real-time or large-scale automated vulnerability mining pipelines.

In BAVul, we process vulnerable and non-vulnerable functions to obtain their fused vector representations μ_1 and μ_2. The refined embeddings S between these two vectors is then computed using the aforementioned distance metrics, yielding N pairs of the form $\langle S_i, \text{Label}_i \rangle$, where $\text{Label}_i \in \{+1, -1\}$. The label is set to $+1$ if the two samples share the same vulnerability and to -1 otherwise.

IV. Experiment Setup

We aim to address the following research questions (RQs):

- **RQ1:** What are the optimal hyperparameters of the model?
- **RQ2:** Compared to existing deep learning-based approaches, how much does BAVul improve detection accuracy and efficiency?
- **RQ3:** Do the individual components of BAVul contribute to the improvement of detection effectiveness?

A. Datasets

After surveying the existing BigVul [22], CVEFixes [23], CrossVul [24], and DiverseVul datasets [25], we selected DiverseVul as the experimental dataset for this study. DiverseVul augments, merges, and deduplicates the other three datasets, making it a more comprehensive and representative collection

for evaluating vulnerability detection performance. The DiverseVul dataset comprises source code from 933 projects, covers 155 CWE categories, includes 523,956 functions, and contains 41,377 vulnerable functions.

We apply the method described in Section III to embed and merge each function code. For a pair of samples sharing the same vulnerability, their embeddings BAE_1 and BAE_2 are represented as $\langle \mathrm{BAE}_1, \mathrm{BAE}_2, +1 \rangle$, otherwise, they are represented as $\langle \mathrm{BAE}_1, \mathrm{BAE}_2, -1 \rangle$. Dataset was divided into 70% for the training set, 15% for the validation set, and 15% for the test set.

B. Environment

Our experiments were conducted on a PC equipped with an i7-10700 CPU and 32 GB of memory. The embeddings of both code sequences and graphs are implemented using PyTorch and Transformer modules. The CFG, AST, and PDG are generated using the Joern tool.

V. Experimental Evaluation

A. Hyperparameter

To identify the optimal model hyperparameters, we train the model for 100 epochs and evaluate its AUC on the validation set every 10 epochs.

Max Length: In CodeBERT, multiple max length options such as 128, 256, and 512 are available. Our analysis of the vulnerable functions in DiverseVul shows that 95% of them have token counts between 200 and 400. Since vulnerability detection is performed at the function level, we set the maximum sequence length to 512 tokens when embedding the code with CodeBERT. This configuration suffices to cover the vast majority of vulnerable functions while minimizing information loss from truncation.

Attention Heads: To determine the optimal number of attention heads for the GAT and bug attention modules, we evaluated configurations with 1, 2, 4, and 8 heads while fixing the embedding dimension at 768. Previous studies have consistently shown that 4 or 8 heads often yield strong classification performance and computational efficiency in similar graph-based models. As illustrated in Figure. 5a, the model achieves the highest vulnerability detection performance when the number of heads is set to 4. Consequently, we adopt 4 heads for both the GAT and bug attention modules in our final architecture.

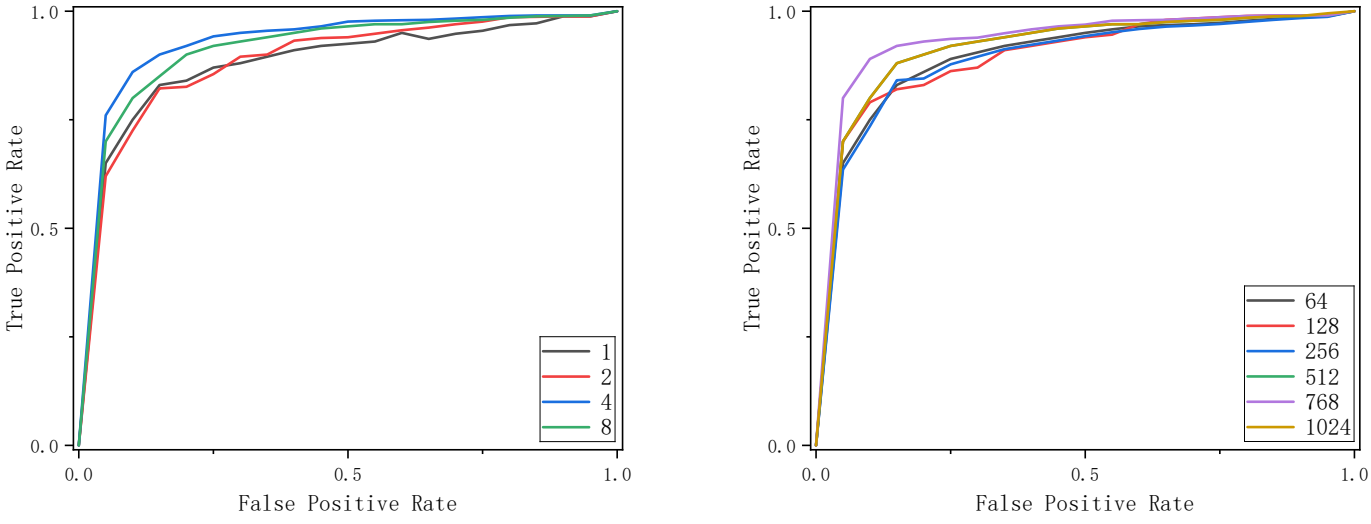
Embedding Size: To determine the optimal embedding size, we investigated five candidate dimensions: 64, 128, 256, 512, and 768 (with an additional consideration of 1024 for completeness). For dimensions not equal to 768, a linear projection layer was appended immediately after CodeBERT to align the sequence embeddings with the graph embedding dimension. The vulnerability detection performance under these configurations is presented in Figure. 5b. the classifier achieves the highest performance when the embedding dimension matches the original 768 dimensional output of CodeBERT. This superior result stems from the ability to fully leverage the pre-trained representations from CodeBERT without dimensionality reduction, thereby preserving the rich, original semantic information of the tokens and enabling more effective semantic enhancement of the graph structure, which ultimately leads to the best detection performance.

> **Answer to RQ1:**After thorough hyperparameter tuning and evaluation, we determined the optimal values for the maximum sequence length, embedding dimension, and number of attention heads. The selected maximum sequence length of 512 tokens suffices to encompass the vast majority of vulnerable functions without incurring substantial information loss from truncation. Meanwhile, extensive validation confirms that the selected embedding dimension and head count yield the highest vulnerability detection performance.

B. Detection Performance comparison

1) Comparision with Other Methods: To evaluate the vulnerability detection performance of BAVul, we compare it with several state-of-the-art(SOTA) deep learning based methods. The five representative approaches included in this comparison are VulSeeker [4], FIT [5], VulSim [6], DeepDFA [7], and AugSliceVul [8]. These methods represent prominent deep learning techniques that leverage diverse graph representations and embedding strategies for software vulnerability detection.

- **VulSeeker** and **FIT** first convert the program into a CFG. They then generate semantically enriched representations, namely labeled semantic flow grap (LSCFG) for VulSeeker and 3-level-features attributed control flow graph(3LACFG) for FIT, by incorporating data and operation attributes extracted from the CFG. These enhanced graph representations are subsequently embedded using natural language processing techniques in combination with Structure2Vec [26] to capture both code semantics and structural information. Finally, a siamese network along with similarity metrics is employed to compute function similarity and perform vulnerability detection.
- **VulSim** introduces a vulnerability detection method that leverages three distinct feature types—semantic, contextual, and syntactic. Each feature type is embedded using a dedicated model tailored to its dimension. Cosine similarity is then computed between embeddings to reinforce the feature representations, thereby improving overall detection performance.
- **DeepDFA** identifies the structural similarity between the message passing mechanism of GNN and data flow propagation along CFG. By injecting variable and data information extracted from the CFG into the GNN learning process, DeepDFA enables the model to learn latent data flow patterns that exist across different programs, thereby facilitating the detection of potential security vulnerabilities.
- **AugSliceVul** constructs an augmented program dependence graph (AUG-PDG) by incorporating five types of data and operation attributes into the traditional PDG.



(a) (b)

Fig. 5: Impact of Different Hyperparameters

It then applies program slicing techniques to select vulnerability-relevant patterns and employs a hybrid loss function to fine-tune the embedding module within CodeBERT, achieving precise vulnerability detection.

Figure 6 illustrates the performance comparison of BAVul with aforementioned methods. Clearly, BAVul exhibits markedly better detection effectiveness. The AUC scores of the five baseline approaches are 89.22%, 89.44%, 90.2%,92.10%, and 93.11%, respectively. On average, BAVul surpasses these methods by 2.87% in AUC.

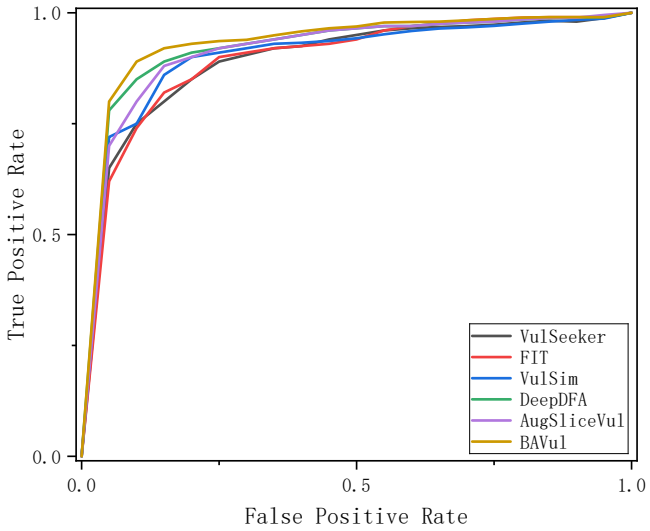


Fig. 6: Comparision with Other Works.

The superior performance of BAVul stems from its effective integration of token level and three types of graph level semantic information via the bug attention mechanism, which substantially enriches the resulting graph representations. In contrast, prior approaches typically focus on enhancing features or models derived from a single source, without adequately addressing the fusion of multiple feature types or their inherent interrelationships. Consequently, BAVul achieves the highest vulnerability detection performance among the evaluated methods.

Table I compares the effectiveness and efficiency of BAVul with the aforementioned baseline methods. VulSeeker and FIT, relying solely on relatively simplistic control flow graph representations, deliver limited detection performance, achieving f1-scores of 76.32% and 81.22%, respectively. In contrast, VulSim, DeepDFA, and AugSliceVul integrate representations from multiple program feature sources and incorporate feature specific modeling strategies (e.g., DeepDFA's simulation of data-flow analysis), resulting in substantially higher detection performance with f1-scores of 89.36%, 92.82%, and 91.67%, respectively. BAVul preserves diverse feature types while explicitly capturing their internal relationships, thereby strengthening multi-feature semantic fusion. Detection performance is further improved through distance-based dimensionality reduction and LightGBM, yielding highest f1-score of 96.85% and accuracy of 96.86%.

TABLE I: Comparison with Other Methods

Method	Accuracy	Precision	Recall	F1-Score	Average Time
VulSeeker	75.99%	75.28%	77.39%	76.32%	0.32min
FIT	81.01%	80.35%	82.11%	81.22%	0.75min
VulSim	89.51%	90.68%	88.08%	89.36%	1min
DeepDFA	92.84%	93.06%	92.58%	92.82%	2min
AugSliceVul	91.74%	92.47%	90.88%	91.67%	3.23min
BAVul	96.86%	97.02%	96.68%	96.85%	1.35min

Regarding efficiency, VulSeeker and FIT leverage simple siamese networks, resulting in moderate runtime advantages with average detection times of 0.32 min and 0.75 min,

respectively, their primary overhead stems from program internal information reconstruction. VulSim benefits from a rule based feature extraction approach and exhibits average detection time of 1 min. DeepDFA incurs longer detection time due to explicit data flow analysis (average 2 min), while AugSliceVul suffers from substantial overhead introduced by program slicing (average 3.23 min). BAVul strikes a balanced trade off between detection performance and computational efficiency. Its primary computational cost stems from the token and graph embedding stages, whereas distance computation and model inference incur negligible overhead, resulting in an average detection time of 1.35 minutes. Compared to data-flow analysis and program slicing-based methods (DeepDFA and AugSliceVul), BAVul demonstrates a clear advantage in both detection effectiveness and runtime efficiency: it reduces average detection time by 32.50% compared to DeepDFA (2 minutes) and by 58.20% compared to AugSliceVul (3.23 minutes).

2) Known Vulnerability Detection: To evaluate BAVul's performance on specific vulnerability categories compared with the aforementioned baselines, we focus on two types of command injection vulnerabilities (CWE-77 and CWE-78) and two types of buffer overflow vulnerabilities (CWE-119 and CWE-120), both of which can cause severe security consequences. As shown in Figure. 7, BAVul achieves the best classification performance on the first two vulnerability types. Across all four categories, BAVul attains detection accuracy exceeding 93%, with a detection accuracy of 99% for CWE-77.

When detecting CWE-119 and CWE-120 vulnerabilities, BAVul achieves slightly lower accuracy than DeepDFA, with differences of 0.99% and 0.36%, respectively. This can be attributed to DeepDFA's explicit modeling of data flow analysis, which provides an advantage in certain vulnerability types that heavily depend on data propagation patterns. However, BAVul exhibits significant advantages in detecting the first two vulnerability types. As evidenced by the results in Table. I, BAVul also demonstrates superior overall detection performance across all datasets. Furthermore, thanks to its feature fusion and refinement modules, BAVul achieves substantially lower detection time compared to DeepDFA.

Answer to RQ2: After extensive testing across multiple baselines, we demonstrate that BAVul leverages the internal relationships among diverse feature types to achieve more precise vulnerability detection than existing approaches. In particular, it effectively identifies two prevalent vulnerability classes (command injection and buffer overflow) while substantially reducing inference latency in the deep learning based vulnerability detection process through feature refinement and efficient machine learning techniques.

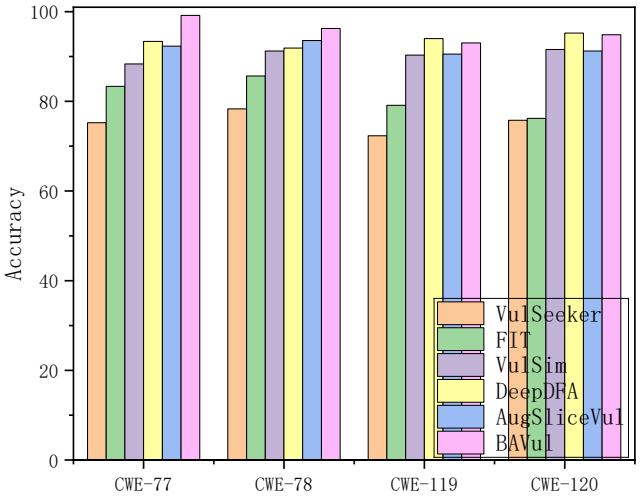


Fig. 7: Specific Type Vulnerability Detection.

VI. Ablation Study

To evaluate the impact of the proposed bug attention mechanism, feature refinement, and the application of bug attention to different feature types in BAVul, we conducted an ablation study by isolating individual components, as reported in Table II. In the baseline configuration without bug attention or feature refinement, token and three graph embeddings (AST, CFG, PDG) are simply concatenated and fed into LightGBM for detection, yielding an F1-score of 92.25%. While this setup demonstrates the benefits of multi-feature fusion, the absence of explicit modeling of internal feature relationships limits overall performance. When bug attention is applied without feature refinement, the F1-score improves to 95.65%, approaching the full BAVul performance. However, the lack of refinement introduces redundancy in the features, leading to overfitting and a modest degradation relative to the complete model.

Applying the bug attention mechanism selectively to different graph structures reveals that the AST yields the highest f1-score of 94.85%, followed closely by the PDG with 94.56%. This superior performance arises from the AST's comprehensive representation of program syntax and semantics, as well as the PDG's clear depiction of program dependencies. In contrast, the CFG primarily captures execution paths and data dependencies, which are often more complex and highly entangled. Consequently, the bug attention mechanism captures these relationships less effectively in the CFG than in the AST or PDG. Overall, BAVul applies bug attention across

TABLE II: Ablation Study Results

Method	Accuracy	Precision	Recall	F1-Score
woRA,woRE	92.38%	93.8%	90.75%	92.25%
withBA,woRE	95.6%	94.68%	96.64%	95.65%
AST+BA+RE	94.86%	95.12%	94.58%	94.85%
CFG+BA+RE	93.31%	94.8%	91.65%	93.2%
PDG+BA+RE	94.62%	95.6%	93.54%	94.56%
BAVul	96.86%	97.02%	96.68%	96.85%

multiple graph representations to capture enriched semantic information, while feature refinement mitigates overfitting and reduces inference latency. This combination effectively integrates the strengths of diverse features, enabling highly accurate and efficient vulnerability detection.

Answer to RQ3: Ablation studies on the key components of BAVul confirm that the bug attention based semantic enrichment strategy significantly outperforms a naive feature concatenation approach (fusion-only baseline). Feature refinement effectively mitigates overfitting induced by high dimensionality, thereby improving both detection efficiency and precision. Furthermore, fusing embeddings from the three graph types (AST, CFG, and PDG) enables the model to capture more comprehensive semantic features, ultimately yielding the highest vulnerability detection performance.

VII. CONCLUSION

In this paper, we propose BAVul, a vulnerability detection system that employs a bug attention mechanism to semantically enrich graph representations of code. The system first processes source code using CodeBERT to obtain token level semantic embeddings. Based on these token semantics, cross attention is applied to enhance the representations of three program graph types (AST, CFG, and PDG). Subsequently, distance based feature refinement and a machine learning classifier are utilized to perform vulnerability detection, effectively mitigating overfitting and reducing computational overhead compared to purely deep learning based methods. This approach substantially outperforms single feature detection techniques in both effectiveness and efficiency.

Future work will improve feature interpretability and effectiveness. Sequence and graph features can be strengthened via heuristic pruning to better capture vulnerability semantics. To meet the demand for large-scale analysis, we will explore model lightweighting techniques that maintain accuracy while enhancing usability and real-time applicability.

REFERENCES

[1] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th annual computer security applications conference*, pp. 481–496, 2022.

[2] Y. Guo, S. Bettaieb, and F. Casino, "A comprehensive analysis on software vulnerability detection datasets: trends, challenges, and road ahead," *International Journal of Information Security*, vol. 23, no. 5, pp. 3311–3327, 2024.

[3] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2365–2376, 2022.

[4] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 896–899, 2018.

[5] H. Liang, Z. Xie, Y. Chen, H. Ning, and J. Wang, "Fit: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching," *Computers & Security*, vol. 99, p. 102032, 2020.

[6] S. Shimmi, A. Rahman, M. Gadde, H. Okhravi, and M. Rahimi, "{VulSim}: Leveraging similarity of {Multi-Dimensional} neighbor embeddings for vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1777–1794, 2024.

[7] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th ieee/acm international conference on software engineering*, pp. 1–13, 2024.

[8] Z. Zou, T. Jiang, Y. Wang, T. Xue, N. Zhang, and J. Luan, "Code vulnerability detection based on augmented program dependency graph and optimized codebert," *Scientific Reports*, vol. 15, no. 1, p. 39301, 2025.

[9] google, "american fuzzy lop," 2026.

[10] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 713–724, 2020.

[11] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–38, 2023.

[12] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, and L. Sun, "Fits: Inferring intermediate taint sources for effective vulnerability analysis of iot device firmware," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pp. 138–152, 2023.

[13] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Xu, Z. Li, P. Di, Y. Jiang, *et al.*, "Llm-powered static binary taint analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–36, 2025.

[14] S. Yang, Z. Xu, Y. Xiao, Z. Lang, W. Tang, Y. Liu, Z. Shi, H. Li, and L. Sun, "Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–29, 2023.

[15] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, pp. 361–374, 2022.

[16] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2526–2540, 2021.

[17] K. Hough and J. Bell, "A practical approach for dynamic taint tracking with control-flow relationships," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–43, 2021.

[18] Z. Feng, "Codebert: A pre-trained model for program-ming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[21] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[22] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th international conference on mining software repositories*, pp. 508–512, 2020.

[23] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 30–39, 2021.

[24] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "Crossvul: a cross-language vulnerability dataset with commit data," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1565–1569, 2021.

[25] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 654–668, 2023.

[26] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, pp. 2702–2711, PMLR, 2016.