Engineering Applications of Artificial Intelligence 177 (2026) 114866

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Research paper

Modeling function-level relationships for vulnerability detection in graph neural networks

Rosmaël Zidane Lekeufack Foulefack *, Elisabetta Provvedini, Alessandro Marchetto

Department of Information Engineering and Computer Science, University of Trento, Via Sommarive 9, Trento, 38123 TN, Italy

ARTICLE INFO

Dataset link: SCVdet

Keywords:
Inter-function dependencies
Vulnerability detection
Deep learning
Software security

ABSTRACT

Context: Deep learning, and in particular graph-based models, has advanced software vulnerability detection by effectively capturing structural code features. Nonetheless, most existing approaches treat source code as independent components, overlooking inter-function relationships and thereby missing potential vulnerability propagation across function boundaries.
Objective: To address this limitation, we propose **SCVdet**, a Graph Attention Network (GAT) based method that integrates inter-function dependencies into the vulnerability detection process.
Method: These dependencies are approximated by using a clustering technique over learned code functions' embeddings. This enables scalable and dynamic modeling of function relationships without incurring the high computational cost of the static code analysis required to recover the full set of real and complete inter-function dependencies. We enhance representation learning by complementing the analysis of the functions' code, local code semantics, with the inter-function dependency analysis, global project-level analysis. We then adopt two strategies: concatenation and attention, to fuse such local and global types of information.
Results: Extensive experiments on multi-language datasets (*FFmpeg+QEMU*, *ProjectKB*, *Big-Vul*, and *CVEFixes*) demonstrate that SCVdet consistently outperforms both sequence-based and graph-based baselines, achieving up to a **16%** improvement in F1-score at the function level and **7%** at the statement level.
Conclusion: The results indicate that SCVdet improves vulnerability detection by combining both local and global information. This approach increases detection accuracy and outperforms the capabilities of state-of-the-art tools while maintaining scalability.

1. Introduction

As modern software systems continue to evolve, the frequency and severity of security incidents have increased (McAfee, 2018; Mitre, 2024). Software vulnerabilities, including hacking, botnet attacks, data breaches, and economic extortion, pose significant threats to software security and the broader software supply chain (securityboulevard, 2025). This underscores the urgent need for robust and scalable software vulnerability detection (SVD) techniques to mitigate security risks effectively.

In recent years, SVD has garnered considerable attention from researchers, leading to the development of diverse detection techniques, primarily targeting source code analysis. These techniques range from traditional static code analysis to contemporary data-driven approaches. Traditional static techniques, often rule-based, have demonstrated some effectiveness in identifying vulnerabilities; they suffer from high false positive rates, limiting their reliability and practical applicability (Croft et al., 2021). Consequently, there has been a paradigm shift toward AI-based approaches. Data-driven techniques are designed to identify weaknesses in source code that malicious actors could exploit, also remotely. These techniques train machine learning models, deep-learning models (DL), and Large Language Models (LLMs) on historical vulnerability data, and then use these trained models to analyze previously unseen code (Yang and Li, 2024; Kaniewski et al., 2025).

Among AI-based methods, two primary modeling paradigms have emerged: sequential models and graph-based models. While sequential models process source code as token sequences, graph-based models represent code as graphs to better capture its structural properties. Notably, graph-based models have shown improved performance in fine-grained, statement-level (e.g., code line) vulnerability detection tasks (Hin et al., 2022; Fu and Tantithamthavorn, 2022). These models typically employ Graph Neural Networks (GNNs) such as Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs) to

* Corresponding author.
E-mail addresses: rz.lekeufack@unitn.it (R.Z. Lekeufack Foulefack), alessandro.marchetto@unitn.it (A. Marchetto).

https://doi.org/10.1016/j.engappai.2026.114866
Received 30 September 2025; Received in revised form 10 April 2026; Accepted 14 April 2026
Available online 19 April 2026

learn from graph-structured representations of source code (Foulefack et al., 2024b; Yang and Li, 2024).

Several SVD methods have adopted these approaches, including FUNDED (Wang et al., 2020), IVDetect (Y. Li et al., 2021), LineVD (Hin et al., 2022), VulGraB (Wang et al., 2023), GraphFVD (Shao et al., 2025), and EnGS2F (Xiao et al., 2024). These models employ sophisticated architectures, such as attention mechanisms and gated GNNs, to enhance detection accuracy. In particular, methods like LineVD, SEDSVD (Dong et al., 2023), and DetectVul (Tran et al., 2025) focus on line-level vulnerability detection in C, C++, and Python code. Although IVDetect introduces surrounding code context, this information is not explicitly integrated into the model's learning process. More recent works, such as VulBG and HGIVul (Song et al., 2023), leverage contextual and behavioral relationships through program slicing, clustering, and hyperedge construction to capture inter-procedural information. Despite these advances, a key limitation persists: these existing models treat each function of the source code in isolation, ignoring the higher-level interconnections among them. Hence, these approaches do not consider function-level dependencies in their learning phase. This is also due to the way in which existing datasets used to train AI models are constructed, i.e., typically at the function level and based on code version changes (i.e., Git commits), which results in disconnected samples during training.

Fig. 1(a) illustrates the extent of this issue by showing how many vulnerable functions in the QEMU and FFmpeg datasets (Zhou et al., 2019) are invoked by functions labeled as non-vulnerable. Such higher-level connectivity is often neglected, yet it can be critical to understanding the vulnerability propagation across a codebase, i.e., the vulnerability within the called function can propagate in the calling function.

Despite promising performance, existing AI-based SVD models fall short in real-world settings (Chakraborty et al., 2021). They tend to rely heavily on local context (e.g., learn vulnerability patterns only within isolated functions) and often fail to model inter-function dependencies and connections that may carry or mask vulnerabilities across function calls. Given that software systems comprise interconnected functions, sometimes distributed across multiple files, existing SVD models lack the ability to incorporate these higher-level relationships during the learning process. Addressing this issue is crucial, as vulnerabilities in a function can propagate through function calls, potentially introducing security risks in other parts of the software.

In this paper, we propose a method for constructing inter-function dependency graphs to capture function-level dependencies and enhance software SVD using deep learning. Motivated by the observation that invoking a vulnerable function within another function can introduce security risks, we address the challenge of modeling such dependencies by constructing a higher-level graph that describes relationships across the entire code project. Directly constructing a complete project-level dependency graph is often impractical due to scalability constraints, incomplete or noisy static call graph extraction, and the high computational cost associated with large-scale software systems. This typically requires external static analysis tools, which introduce additional time, space, and scalability overheads. To overcome these challenges, we *approximate* inter-function dependencies using the DBSCAN clustering algorithm. This strategy allows the model to process graph representations of individual functions while embedding them within a broader, evolving dependency context. By incorporating these approximated inter-function dependencies into the model's attention mechanism, we aim to facilitate learning from vulnerability propagation patterns while preserving scalability. Unlike prior inter-function approaches, our method does not rely on explicit call graph construction; instead, it learns function-level dependencies directly in the embedding space, making it scalable across large and heterogeneous codebases. We hypothesize that this integration enables GNNs to better capture complex software structures and dynamically model long-range dependencies.

The main contributions of this work are:

- The integration of approximated inter-function dependency information into a GNN-based SVD model, enabling dynamic modeling of vulnerability propagation patterns across functions. We name our proposed model **SCVdet**.
- The design of a scalable approximation strategy based on clustering to construct project-level inter-function dependency graphs without requiring explicit call graph extraction.
- An extensive evaluation on multi-language datasets (*FFmpeg+QEMU*, *ProjectKB*, *Big-Vul*, and *CVEFixes*) in which SCVdet achieves consistent improvements over baselines. Notably, we observe an increase of up to **16%** in F1-score at the function level and **7%** at the statement level.

The remainder of this paper is organized as follows: Section 2 details the motivation. Section 3 presents our proposed method. Section 4 describes our experimental setup, and Section 5 discusses the results. Section 6 provides further analysis and discussion. Section 7 reviews the related work, and Section 8 concludes the paper and outlines future work.

2. Motivation

A vulnerability is a weakness in source code that can be exploited by cyber attackers to gain unauthorized access to software systems. Such vulnerabilities may be exploited through techniques such as SQL injection, buffer overflows, cross-site scripting (XSS), or by automated exploit kits that scan for known weaknesses in applications. As modern software systems grow increasingly complex, the early detection of vulnerabilities becomes a critical research challenge.

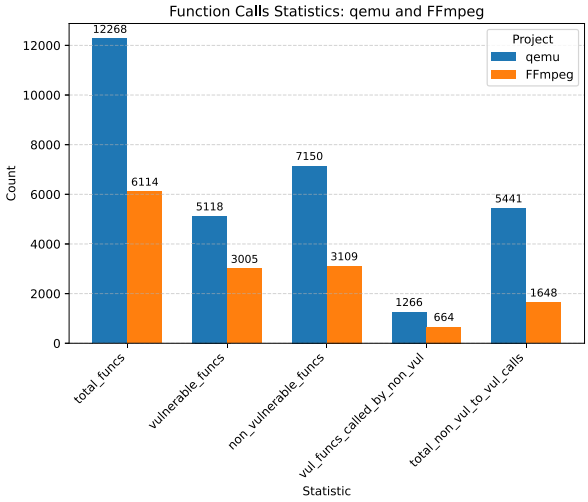
Despite considerable advancements of AI-based vulnerability detection methods, a fundamental limitation persists in most of the existing methods: they analyze functions in isolation and overlook dependencies between them, such as function calls or data-flow dependencies.

Most prior work processes code at the function or snippet level, assuming each function to be an independent unit. For example, several studies (Fu and Tantithamthavorn, 2022; Hin et al., 2022; Mirsky et al., 2023; Tran et al., 2025; Liu et al., 2024; Dong et al., 2023) have introduced models that classify code vulnerabilities based on individual functions or statements. Although Y. Li et al. (2021) define the vulnerability context as including surrounding code that is control and/or data dependent within the same function, their learning approach still processes each function independently. While these approaches have demonstrated promising results, they lack the capacity to capture how vulnerabilities may propagate through function calls across the code of a software system.

Fig. 1(b) illustrates a simplified example: function `func1` is known to be vulnerable, and function `func2` invokes `func1` at line 5. Although `func2` is not directly labeled as vulnerable, it may inherit the security risks of `func1` through this call. Traditional AI-based models that treat functions independently are likely to misclassify such cases, as they are unaware of inter-functional influence. In fact, from the perspective of `func2`, line 5 might appear safe despite encapsulating behavior from a vulnerable source.

This phenomenon is not merely theoretical. Fig. 1(a) presents empirical evidence from Qemu and FFmpeg. In the Qemu dataset, 1266 vulnerable functions (out of 12 268 total functions) are invoked by functions that are not labeled as vulnerable. Similarly, in FFmpeg, this occurs 664 times (out of 6114 total functions). Additionally, we report 17 such functions in QEMU and 41 in FFmpeg that are inconsistently labeled as both vulnerable and non-vulnerable. These statistics highlight the practical importance of modeling function-level dependencies.

Modern software is made up of many interconnected functions spread across files and modules. These dependencies greatly impact an application's security posture. Overlooking them reduces the effectiveness of AI-based detection models, especially in identifying vulnerabilities that cross functional boundaries.



(a) Statistics of vulnerable functions invoked by non-vulnerable functions in the QEMU and FFmpeg datasets identified by the presence of token names.

```
1 def func1(user_input):
2     # Vulnerable function
3     query = "SELECT * FROM users WHERE name = '" + user_input + "';"
4     print("Executing query:", query)
5     return query

1 def func2():
2  # Function that calls func1
3  # user_data = input("Enter username: ")
4     user_data = "default_user"
5     result = func1(user_data)  # (potential vulnerability propagation)
6     print("Query result:", result)
7     return result
```

(b) Example of potential vulnerability propagation through function calls.

Fig. 1. Statistics on function calls and an example of vulnerability propagation.

We hypothesize that integrating function-level dependencies into the learning process can substantially enhance the detection capabilities of AI models. By allowing the model to attend not only to the local context within a function but also to its interactions with other functions, we can better capture vulnerability propagation patterns. This hypothesis is partially supported by recent related findings (Marchetto and Lekeufack Foulefack, 2024), which emphasize the benefits of context-aware analysis in source code learning.

3. Methodology

This research presents a novel graph-based approach to model and evaluate function-level dependencies for vulnerability detection in source code.

3.1. Problem formulation

Graph Neural Networks have shown strong capabilities in extracting insights from structured data like source code. They work by passing messages between nodes in a graph to learn both local and global contextual features.

In this work, we treat statement-level (code line) vulnerability detection as a binary node classification problem. Each data instance is a tuple (G_i, V_i, Y_i), where G_i is the graph, V_i is a node representing a program statement, and $Y_i \in 0, 1$ is its label; 1 for vulnerable, and 0 for non-vulnerable.

A GNN model f is trained to map $V \rightarrow Y$ by minimizing cross-entropy loss. Due to the imbalance in certain datasets, specifically in the node label distribution, we apply a dual-objective loss function. This function combines a conditional node-level loss with a graph (function)-level loss, ensuring the model learns both localized vulnerability patterns (i.e., node level) and the overall function's security property. An entropy regularization term (λ) is added to prevent overconfidence in the dominant negative class. Importantly, the node-level loss is applied conditionally: it contributes to the total loss only when a graph contains at least one vulnerable node. This selective supervision strategy ensures that learning focuses on the class of interest, the vulnerable code, while reducing gradient noise from fully non-vulnerable functions.

Additionally, we condition the node-level loss on the presence of vulnerable nodes to improve the model's discriminative capacity. This approach emphasizes informative samples, which helps prevent the model from simply classifying all nodes as non-vulnerable. On the other hand, the function-level loss offers global contextual guidance, ensuring the model maintains a comprehensive view of the global function-level security. The final loss $loss\_fn$ for each training sample is defined as a weighted combination of three components:

- **Conditional Node Classification Loss** ($\mathcal{L}_{\text{node}}$): A cross-entropy loss over node labels, applied only to graphs containing at least one vulnerable node.
- **Function Classification Loss** ($\mathcal{L}_{\text{func}}$): A cross-entropy loss for graph-level classification, distinguishing vulnerable from non-vulnerable functions.
- **Entropy Regularization Term** (λ): A regularization term that promotes prediction uncertainty at the node level, improving generalization and compensating for cases where the node-level loss is inactive.

Mathematically, the formulation of this approach is as follows:

Let a graph G consist of a set of nodes V. Let $\mathbf{z}_{\text{node}}$ denote the node-level prediction logits and $\mathbf{z}_{\text{func}}$ the graph-level prediction logits. The corresponding ground-truth labels are given by the node label vector y_{node} and the graph-level label y_{func}.

We define an indicator function $\mathbb{I}_{\text{vuln}}(G)$, which equals 1 if the graph G contains at least one vulnerable node and 0 otherwise.

The loss components are defined as follows:

$$\mathcal{L}_{\text{node}} = \mathbb{I}_{\text{vuln}}(G) \cdot \text{CE}(\mathbf{z}_{\text{node}}, y_{\text{node}}) \tag{1}$$

$$\mathcal{L}_{\text{func}} = \text{CE}(\mathbf{z}_{\text{func}}, y_{\text{func}}) \tag{2}$$

where $\text{CE}(\cdot)$ denotes the cross-entropy loss. The total training loss is defined as:

$$\begin{aligned} loss\_fn &= \mathcal{L}_{\text{total}} \\ &= \mathcal{L}_{\text{node}} + \tfrac{1}{2}\mathcal{L}_{\text{func}} + \lambda \end{aligned} \tag{3}$$

In this case, the graph-level feature refers to the conversion of the source code text utilized to construct the graph. Meanwhile, the node-level feature pertains to the corresponding code line vector representation obtained through code embedding. Algorithm 1 describes the overall learning framework. In Algorithm 1, the input is a graph dataset $D = \{(G_i, V_i, Y_i)\}$, where each graph G_i is associated with a

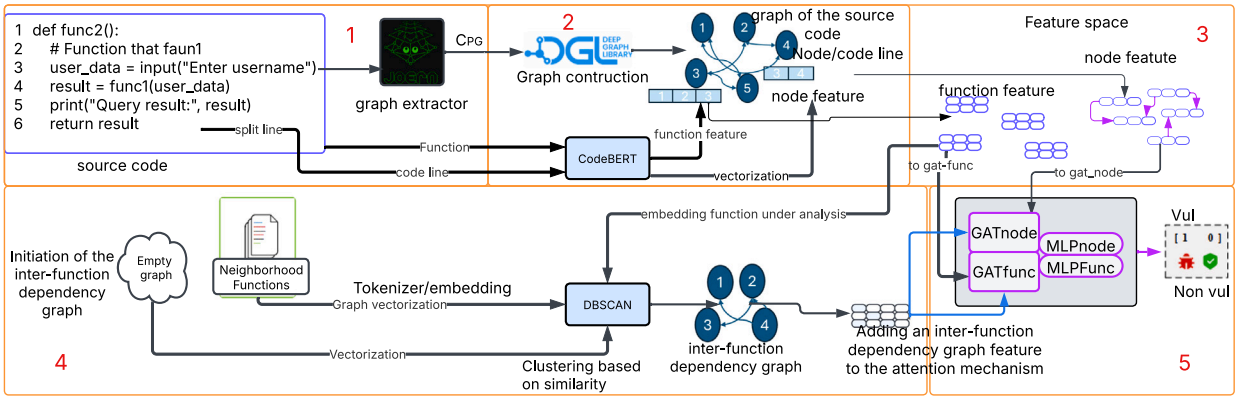


Fig. 2. Overview of the proposed method's architecture for identifying vulnerabilities in source code.

set of nodes V_i and corresponding node-level labels Y_i. The goal is to learn a model f, parameterized by θ, that classifies each node in V_i. Training proceeds for T epochs with learning rate η. At each epoch, the dataset is shuffled and divided into mini-batches $\mathcal{B}$ (small set of training data). For each mini-batch $(feature, label) = (x, y) \in \mathcal{B}$, the model generates predictions *(line 5)*. The training objective is the learnable cross-entropy loss *(line 6)*, where $C = 2$ in the binary vulnerability detection task. Subsequently, the model parameters are updated via gradient descent *(line 7)*, by iteratively minimizing the loss, the model improves its ability to classify nodes as vulnerable or non-vulnerable.

Algorithm 1 Node Classification for Vulnerability Detection

Require: Graph dataset $\mathcal{D} = (G_i, V_i, Y_i)$, learning rate η, number of epochs T
Ensure: Trained model $f : V \to Y$
1: Initialize model parameters θ
2: **for** $t = 1$ to T **do**
3: Shuffle $\mathcal{D}$ and create mini-batches $\mathcal{B}$
4: **for all** mini-batch $(x, y) \in \mathcal{B}$ **do**
5: Compute predictions $\hat{y} = f(G_i, V_i; \theta)$
6: Compute learnable cross-entropy loss $loss\_fn$ (3):

$$\text{where } \mathrm{CE}(\cdot) = \mathcal{L}(y, \hat{y}) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c), \quad C = 2$$

7: Update model parameters: $\theta \leftarrow \theta - \eta \nabla_\theta L$
8: **end for**
9: **end for**

3.2. System architecture

3.2.1. Graph data extraction

Graph data extraction (Fig. 2, step 1) is the first crucial step in constructing a representation of source code suitable for graph-based learning models. In the literature, various tools and techniques have been developed for this purpose, including `AST` modules in Python, `PHP-Parser` for PHP, `Soot` for Java, `ANTLR` for general parsing, and `Joern` for low-level code analysis. Among these, Joern (2024) has gained wide adoption due to its capability to analyze multiple programming languages and generate rich code representations, such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Dependency Graphs (DDGs), and Program Dependency Graphs (PDGs). These representations facilitate the construction of comprehensive graph structures that model both the syntactic and semantic aspects of the code. In this work, we use `Joern` to extract such representations and construct graphs where nodes represent program elements (e.g., code lines and statements) and edges denote various types of syntactic or semantic relationships (e.g., data dependency-edge connecting a variable definition to its latter usage).

3.2.2. Graph feature generation

Graph feature generation (still Fig. 2, step 2) involves transforming the raw source code (e.g., the code from each function) into meaningful vector representations that can be assigned to nodes and edges of the graph. These features play a critical role in enabling learning models to capture and differentiate program semantics. Graph features can be derived in two ways: by using the vector representation of each function as node features for function-level detection or by aggregating line-level vectors (e.g., mean of statement features) when nodes represent individual code lines. In Step 2 of Fig. 2, the source code is decomposed into individual statements, which are parsed and encoded by the source code vectorizer (CodeBERT) to obtain node-level feature representations. In parallel, the complete function body is processed to extract features that capture graph-level (function-level) semantics. These representations are more clearly illustrated in the feature space shown in Step 3, where statement-level features are interconnected according to the CPG edges, resulting in a graph representation for each function.

In contrast, the graph-level feature vectors representing entire functions are initially independent and do not exhibit explicit connections. In the subsequent stage (Step 4), function-level features are used to create an inter-function dependency graph. This graph models the dependencies between functions and facilitates the propagation of global contextual information.

Several embedding models have been proposed for representing source code, including `Word2Vec` (Mikolov et al., 2013), `Sentence-BERT` (Reimers and Gurevych, 2019), and `CodeBERT` (Z. Feng et al., 2020). While `Word2Vec` has been widely adopted in software engineering tasks, the impact of different embedding choices on graph-based vulnerability detection remains underexplored, particularly for transformer-based models such as `Sentence-BERT` and `CodeBERT`. This motivates our comparative evaluation of embedding techniques.

- **Word2Vec** is a classic word embedding technique that learns vector representations of words by leveraging their co-occurrence statistics in a corpus. Despite its simplicity, Word2Vec remains effective in capturing syntactic and semantic relationships between words and is widely used in natural language processing tasks, including source code analysis.
- **Sentence-BERT (SBERT)** is a modification of the BERT architecture that enables the generation of semantically meaningful sentence embeddings. It employs Siamese and triplet networks to produce fixed-size vector representations of sentences, which are well-suited for tasks involving similarity comparison and semantic retrieval using measures like cosine similarity.
- **CodeBERT** is a transformer-based pre-trained model designed for both natural language and programming language processing. It

is trained on paired natural language and code data across six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go), making it particularly suited for tasks that involve code understanding and representation.

3.2.3. Dependency graph construction

Unlike existing approaches, our method constructs an inter-function dependency graph to capture functional relationships between code segments (Fig. 2, step 4). This design enables the approximation of high-level calls and semantic associations based on structural and behavioral patterns.

The graph is constructed using unsupervised learning and similarity analysis in the feature space (step 3). Each function-level graph is treated as a node in a higher-level inter-function graph, where edges are formed by combining clustering and semantic similarity to approximate potential functional interactions. These similarities are presented in Fig. 2, step 4, as those in the same cluster as the entity neighborhood function.

The process comprises the following steps:

(1) **Embedding:** Each function graph g_f is encoded into a fixed-size embedding vector e_f that captures its structural and contextual information. We use two complementary methods for graph embeddings:

- *GraphSAGE* (Hamilton et al., 2017): this aggregates neighborhood information to generate node embeddings and generalizes well to unseen graphs.
- *Node2Vec* (Grover and Leskovec, 2016): which learns embeddings by simulating biased random walks, preserving both local and global graph topology.

(2) **Clustering:** The set of graph embeddings $\{e_f\}$ is clustered using the *DBSCAN* algorithm (Density-Based Spatial Clustering of Applications with Noise). *DBSCAN* is particularly suitable here due to its ability to detect arbitrarily shaped clusters and its independence from a predefined number of clusters. In contrast, methods like *k-means* (Jin and Han, 2010) assume spherical clusters and require pre-specification of the cluster count, making them less flexible for our use case.

The motivation for clustering is to identify groups of semantically and structurally similar functions, which likely share contextual or calling relationships. By means of clustering, we aim to approximate the inference of the functions' dependencies. Unlike static call graphs that depict explicit invocation relationships, embedding-based clustering captures latent behavioral similarities between functions. Such similarity can reflect indirect dependencies (e.g., common source code patterns) that are not observable through direct calls alone.

(3) **Edge Construction:** Within each cluster, we compute pairwise cosine similarity between embedding vectors. An edge is added between two function graphs if their similarity exceeds a predefined threshold δ (empirically defined), ensuring that connections reflect meaningful semantic similarity, not just cluster proximity. The resulting graph encodes potential inter-function dependencies by linking functions that are both structurally and semantically related.

This dependency graph G_L is then integrated into the overall model in step 5. Specifically, its vector representation is used to compute attention scores (4) and concatenated to existing features, which are fused with the features of the isolated function-level graph during training. This fusion allows the model to benefit from global context when making both function and line-level vulnerability predictions. Fig. 3 shows how this integration occurs. Mechanism 1 performs a direct fusion by concatenating the dependency graph representation with the existing node features (i.e., source code embeddings h). In contrast, Mechanism 2 performs an attention-based fusion, where a dot-product between h and the dependency graph representation is used to compute a weighting factor that modulates the original features before aggregation, as defined in (5). The complete procedure for constructing the inter-function dependency graph is detailed in Algorithm 2.

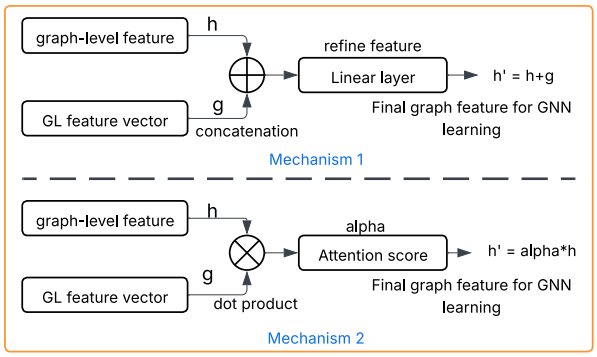


Fig. 3. Integration of constructed dependency information in the process.

The process aims to construct a function-level graph G_L using feature representations of the source code graphs g_i within a mini-batch (cluster C). Each graph g_i is incorporated such that the resulting dependency graph contains only the nodes corresponding to the code samples from which the model learns. We used embeddings derived from the source code graphs to ensure that the vector representations do not introduce programming-specific artifacts (i.e., characters). Such artifacts could distort the vector embeddings by generating disproportionately high values, which could compromise the distance threshold δ and often lead to truncation. These vectors e_f are obtained using GraphSAGE, after which clustering is performed with DBSCAN. A new node corresponding to g_i is added to G_L if its distance from existing neighbors in the embedding space exceeds the threshold δ.

Algorithm 2 Constructing the Inter-function Dependency Graph G_L

Require: Function-level graphs $G_f = \{g_1, g_2, \ldots, g_n\}$, similarity threshold δ
Ensure: Inter-function dependency graph G_L

$G_L \leftarrow$ EmptyGraph()
for all $g_f \in G_f$ **do**
  $e_f \leftarrow$ GraphEmbedding(g_f)
  AddNode(G_L, g_f, e_f)
end for
$C \leftarrow$ DBSCAN($\{e_f\}$) ▷ Cluster embeddings
for all cluster $c \in C$ **do**
  for all pairs (g_i, g_j) in c **do**
    $s \leftarrow$ CosineSimilarity(e_i, e_j)
    if $s > \delta$ **then**
      AddEdge(G_L, g_i, g_j, s)
    end if
  end for
end for
return G_L

3.2.4. Source code classification

In the final stage (Fig. 2, step 5), two independent Graph Attention (GAT) layers are employed to process the features extracted at both the function and node levels. This component performs statement-level vulnerability classification by leveraging the enriched graph representations constructed in the preceding steps. The model integrates both local (node-level) and global (inter-function) information to construct a discriminative embedding space suitable for classification.

Let $G = (V, E)$ denote the input graph, where V is the set of nodes representing program statements (line of code), and E is the set of edges capturing syntactic and semantic dependencies. Each node $v_i \in V$ is associated with a feature vector $h_i \in \mathbb{R}^d$. To incorporate inter-function dependencies, we define an additional graph-level representation vector $g \in \mathbb{R}^d$, which encodes global functional dependencies derived from the function-level connectivity graph.

An attention mechanism is used to dynamically compute the importance of each node's feature in relation to the global context.

Table 1
Architecture details of the proposed **SCVdet** model.

Component	Layer type	Details/Activation
Feature Projections		
rand_proj	Linear	Random feature projection
femb_proj	Linear	Function embedding projection
emb_proj	Linear	Code (line) embedding projection
node_comb_proj	Linear	Combine random + code features
func_comb_proj	Linear	Combine random + function features
Node-Level GAT Layers		
gat1_node	GATConv	Dropout(0.4), LeakyReLU(0.2)
gat2_node	GATConv	Dropout(0.4), LeakyReLU(0.2)
Function-Level GAT Layers		
gat1_func	GATConv	Dropout(0.4), LeakyReLU(0.2)
gat2_func	GATConv	Dropout(0.4), LeakyReLU(0.2)
Node Classification MLP		
node_mlp[0]	Linear	ReLU, Dropout(0.4)
node_mlp[1]	Linear	ReLU, Dropout(0.4)
node_mlp[2]	Linear	Output logits
Function Classification MLP		
graph_mlp[0]	Linear	Output logits
Auxiliary Components		
v1_projector	Linear	Projection of global dependency vector (Use in GAT layers)
beta_mlp[0]	Linear	Sigmoid (attention scores using a global dependency vector)
loss_fn	LearnableWeightedLoss	Custom weighted loss with entropy regularization

Specifically, for each node v_i, we compute an attention score α_i as the dot product between its feature vector h_i and the global representation g. We adopt a multi-head attention mechanism. Each attention head learns an independent projection. For each node v_i and attention head $k \in \{1, \ldots, K\}$, we first compute head-specific linear projections:

$$h_i^{(k)} = W^{(k)} h_i, \quad g^{(k)} = U^{(k)} g,$$

where $W^{(k)}$ and $U^{(k)}$ are learnable projection matrices for the kth head.

The attention coefficient for node v_i under head k is then computed as:

$$\alpha_i^{(k)} = \frac{\exp\left((h_i^{(k)})^\top g^{(k)}\right)}{\sum_{j \in \mathcal{N}(i)} \exp\left((h_j^{(k)})^\top g^{(k)}\right)}, \tag{4}$$

where $\mathcal{N}(i)$ denotes the neighborhood of node i (including itself), and the softmax operation normalizes the attention scores within each head.

The updated representation for node v_i is then computed as a weighted sum of the features of its neighbors:

$$h_i'^{(k)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_j^{(k)} \cdot h_j^{(k)}\right), \tag{5}$$

where $\sigma(\cdot)$ denotes a non-linear activation function (e.g., ReLU).

The output embeddings from the two GAT layers are passed through a linear layer of a Multi-Layer Perceptron (MLP), which produces the vulnerability prediction for each node:

$$\hat{y}_i = \text{MLP}(h_i'). \tag{6}$$

This mechanism enables the model to dynamically attend to the most relevant code regions while jointly capturing both local and global semantic information.

3.2.5. Overall architecture

We recall that Fig. 2 summarizes the overall architecture of the presented SVD method. Instead, Table 1 details the model architecture. In our architecture, `gat_node` is responsible for processing line-level (statement) features, whereas `gat_func` operates on function-level representations. Feature fusion further incorporates normally distributed random features (rand_feat), which act as a regularization mechanism to mitigate overfitting and enhance robustness against adversarial perturbations of the input data.

The proposed model adopts a dual-pathway architecture aligned with the dual-loss formulation, explicitly separating statement-level and function-level vulnerability detection. The statement pathway processes code embeddings through dedicated GAT layers (`gat1_node`, `gat2_node`) to enable fine-grained, line-level classification. In parallel, the function pathway employs independent GAT layers (`gat1_func`, `gat2_func`) to capture holistic, function-level vulnerability characteristics.

To preserve task-specific learning and avoid representational interference, information sharing between the two pathways is deliberately constrained. Only minimal foundational features (denoted as `rand_feat`) are shared, ensuring that each pathway remains focused on its respective objective. Global dependency information is injected into both pathways through a controlled attention/concatenation-based mechanism, allowing global contextual patterns to guide local predictions without introducing cross-task contamination.

Then, independent MLP heads are tailored to each classification granularity. The entire system is trained collaboratively using a unified loss formulation, enabling both pathways to complement each other while maintaining clear functional separation.

The first column of Table 1 refers to the model components, subdivided into six parts. The **Feature Projection** component ensures that, regardless of the embedding tool employed (e.g., CodeBERT, Word2Vec, or SBERT), the model layers within this component align the input features to the appropriate dimensionality required by subsequent modules, namely the **Node-Level GAT Layers** and the **Function-Level GAT Layers** equipped with dropout. The **Node Classification MLP** and **Function Classification MLP** correspond to multilayer perceptrons that perform the final classification once the features have been processed by the GAT layers. The last component, referred to as the **Auxiliary Component**, refines and updates graph and node features within the GAT layers by incorporating information derived from a dynamically constructed dependency graph. Specifically, two distinct linear layers are employed: one for feature concatenation projection and the other for attention score computation. The `v1_projector` learns a graph-level dependency representation vector, which is subsequently concatenated with the independent features learned by each GAT layer. The beta_mlp[0] then predicts an attention coefficient from

this interdependency representation. This coefficient is used to dynamically weight (via element-wise multiplication) the output features of the GAT layers during the attention computation. Throughout this process, the optimization objective is governed by a learnable loss function, formally denoted as `loss_fn` as defined in Eq. (3). The second column of the table specifies the type of layer used in each component, while the third column provides details on the processed information, indicating whether an input/output state.

4. Experimental design

4.1. Research questions

To evaluate the effectiveness of our proposed method for vulnerability detection, we investigate the following research questions (RQs):

RQ1: *Can clustering-based approximation in the embedding space effectively model inter-function dependencies in project call graphs?* We examine whether the proposed clustering approach effectiveness captures code function-level dependencies and contributes to more informative dependency graphs that can effectively improve vulnerability detection performance.

RQ2: *How does incorporating inter-function dependencies influence the performance of GNN-based models for function and statement-level vulnerability detection?* We aim to assess how integrating of inter-function dependencies can improve the vulnerability detection capabilities of GNN-based models.

RQ3: *To what extent do different code embedding techniques (e.g., CodeBERT, SentenceBERT, Word2Vec) impact the node representations in vulnerability detection tasks?* This question explores the influence of various feature extraction strategies on the effectiveness of node representations.

RQ4: *How does SCVdet improve source code vulnerability detection compared to its counterparts?* In this research question, we aim to evaluate the effectiveness of the proposed detection model against state-of-the-art approaches.

4.2. Datasets

We use four publicly available datasets for evaluation: *FFmpeg+QEMU* (Zhou et al., 2019), *ProjectKB* (Ponta et al., 2019), *Big-Vul* (Fan et al., 2020), and *CVEFixes* (Bhandari et al., 2021). Their key statistics are presented in Table 2.

FFmpeg+QEMU and *Big-Vul* are widely used, real-world datasets, respectively with balanced and imbalanced samples containing vulnerabilities in C/C++ code. *ProjectKB* and *CVEFixes*, on the other hand, are an imbalanced dataset drawn from real-world Java and Python projects, respectively, and are comparatively less explored in the literature.

For each dataset, the preprocessing consists of removing all non-executable elements, such as code comments, and identifying and eliminating duplicate elements. Additionally, a limited number of code functions that cannot be parsed using Joern (version 2.0.3) during the graph extraction phase are excluded. These parsing failures make the graph construction impossible, thereby limiting the potential for a complete graph representation of these specific code samples.

Although datasets were initially labeled at the function level, we extend prior work to derive statement-level (node in our graph-based code representation) labels. Specifically: (1) statements removed in commits that fix vulnerabilities are treated as indicative of vulnerabilities, and (2) statements that exhibit control or data dependencies on the newly added statements in these commits are also marked as related to the vulnerability. This approach is based on the rationale that added statements in such commits are meant to resolve the vulnerability, implying that surrounding or dependent statements, though not directly modified, are still relevant to the vulnerable logic.

Table 2
Dataset summary.

Dataset	Language	Samples	Vul. rate (%)
FFmpeg+QEMU	C/C++	27 318	45.56
ProjectKB	Java	20 155	13.64
Big-Vul	C/C++	188 415	6.13
CVEFixes	Python	19 424	14.86

In the experimental setting, each dataset is partitioned into training, validation, and testing sets using an 80:10:10 ratio to facilitate a reliable and methodologically sound evaluation. This approach allowed for a direct comparison with other existing methods.

4.3. Evaluation metrics

We assess the performance of our model in distinguishing vulnerable code fragments from non-vulnerable ones using a set of well-established evaluation metrics. These metrics are suited also for imbalanced datasets and provide insight into both the accuracy and robustness of the model's predictions. The selected metrics include precision, recall, and F1-score.

- **Precision** – Measures the proportion of correctly predicted vulnerable instances among all predicted vulnerable instances:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7}$$

- **Recall** – Measures the proportion of actual vulnerable instances correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{8}$$

- **F1-score** – The harmonic mean of precision and recall:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{9}$$

where, **TP** (True Positives) refers to correctly predicted vulnerable samples, **FP** (False Positives) are non-vulnerable samples misclassified as vulnerable, and **FN** (False Negatives) are vulnerable samples incorrectly predicted as non-vulnerable.

4.4. Implementation and parameter selection

Dependencies: Our implementation is developed in Python using several key libraries: Joern version 2.0.3 for parsing code, Python 3.10, PyTorch 2.3.1+cu121, and a compatible DGL 2.0.0+cu121 with CUDA for graph learning, NetworkX 2.6.3 for graph operations, the Hugging Face Transformers library 4.42.3 for embedding generation, Ray Tune version 2.32.0 for hyperparameter tuning, scikit-learn 1.4.1.post1 for metric evaluation and clustering computation, and substantial core libraries such as NumPy 1.23.0 and Pandas 1.3.5 for data management.

Hardware Setup: All experiments were run on a local server equipped with an AMD Ryzen 9 7950X 16-core CPU, 128 GB DDR5 RAM, and an NVIDIA RTX 4090 GPU with 24 GB of memory.

Hyperparameter Selection: We use the Ray Tune (Liaw et al., 2018) framework to perform a grid search over model hyperparameters. The F1 score on the validation set serves as the guiding metric. We also apply early stopping: training is halted if validation F1 does not improve for 10 consecutive epochs, reducing the risk of overfitting. In our implementation, the entropy regularization coefficient λ is empirically set to 0.01.

Table 3 summarizes the best hyperparameters obtained in the search space and dependency graph configurations.

Table 3
Hyperparameter configuration grid of the proposed model.

Model parameters		Dependency graph parameters	
Parameter	Values	Parameter	Values
Embedding method	CodeBERT, SentenceBERT, Word2Vec	Graph-to-vector method	GraphSAGE
Input feature size	768, 384, 100	Clustering method	DBSCAN, KMeans
Hidden feature size	[525]	Graph-level fusion method	Attention, Concatenation
Dropout rate	[0.4]		
Learning rate	[1e−5]		
Attention heads	[4]		
Max epochs	50		
Early stopping monitor	val_f1_node		

4.5. Process and baselines

The experiment followed these key steps for each dataset:

1. Preprocessing the dataset to clean (e.g., remove duplicated and partial entries) and split it as 80:10:10 (see Section 4.2).
2. Hyperparameter tuning (see Section 4.4) and training of initial models and baselines.
3. Integration of inter-function dependencies into the SCVdet architecture.
4. Re-execution of the models training (from step 2), by considering different code embedding techniques (see Section 3.2.2).
5. Analysis of results based on all collected evaluation metrics (see Section 4.3).

Each model configuration has been executed 5 times to capture the average trend.

We compare our method against state-of-the-art models supported by existing tools that provide exploitable source code for promoting the experiment replicability. However, not all baselines have been designed for working at both levels of granularity detection (i.e., functions and statements), so we consider them for their designed use. Table 4 summarizes the characteristics of these baselines.

- **VulBG** (Yuan et al., 2023) is a linear-based model incorporating inter-function dependencies through a behavioral graph construction process. To the best of our knowledge, the tool supports the SVD method closest to ours. In fact, it applies K-Means clustering to group similar behavioral patterns (code similarity in embedding space) from sliced code functions and uses the Node2Vec algorithm to convert the behavioral graph into a vector. This approach enables the model to capture the functional relationship between code elements for function-level vulnerability prediction. The tool works at the function-level for the C/C++ programming language. In contrast, **SCVdet** is a GNN, supporting both function-level and line-level vulnerability detection. Instead of relying on static program slicing, which risks discarding critical vulnerability propagation edges across functions (e.g., when a vulnerability flows through helper functions omitted by the slicing process), SCVdet clusters learned function embeddings to approximate inter-function dependencies. This clustering is then used to construct project-wide dependency graphs (which are not documented in VulBG), capturing latent structural and semantic relationships without requiring exhaustive static analysis. Differently, from VulBG, SCVdet trains on these graphs using mini-batch updates, as detailed in our methodology, whereas the update procedure of VulBG is not documented. The additional support for line-level detection also enhances the interpretability of SCVdet's predictions by allowing developers to pinpoint specific vulnerable statements, thereby providing finer-grained insights beyond function-level alerts.
- **AMPLE** (Wen et al., 2023) is a vulnerability detection framework that works at function-level for C/C++. It uses graph simplification to reduce node distances and improve graph representation learning. It incorporates an edge-aware GCN to fuse different edge information and a kernel-scaled module to better understand long-range node relationships.
- **ASTGRU** (H. Feng et al., 2020) is a neural model that leverages the structural information of Abstract Syntax Trees (ASTs) and uses bidirectional Gated Recurrent Units (Bi-GRUs) to model sequential patterns within the code. It is designed to capture both syntactic and semantic information at the function level. The tool works at function-level.
- **JLineVD** (Foulefack et al., 2024a) is an enhanced version of the LineVD (Hin et al., 2022) model that works at both function and statement level, specifically adapted for Java source code. By using Node2vec, it integrates contextualized node and edge embeddings to better capture long-range dependencies during training, thereby improving statement-level vulnerability prediction.
- **IVDetect** (Y. Li et al., 2021) is a graph-based model specifically fine-tuned for detecting source code vulnerabilities as a binary classification task. It extracts control and program dependency graphs to construct subgraphs for statement-level vulnerability detection. The tool works at function level for both C/C++ and Java.
- **SVDet** (Marchetto and Lekeufack Foulefack, 2024) extends the VDet framework by incorporating interpretability through the SHAP (SHapley Additive exPlanations) method. This enhancement allows for more transparent decision-making in the vulnerability detection process and supports statement-level predictions in Java code.
- **ANGEL** (Peng et al., 2025) performs vulnerability detection on C/C++ source code using a combination of graph transformers and GNNs. The method simplifies the CPGs to reduce redundancy and noise, enabling more effective function-level vulnerability identification.
- **RoS-Dex** (Do Xuan et al., 2026) is an ensemble-based approach designed to detect vulnerabilities in C/C++ programs. It incorporates a local rotary positional embedding to enrich positional information in the embeddings, helping preserve the structural layout of the source code.
- **MLAF-VD** (Li et al., 2025) extracts multi-level abstract code features through multiple attention mechanisms. This design allows the model to capture global graph representations that better reflect the underlying program semantics.
- **LineVul** (Fu and Tantithamthavorn, 2022) is a transformer-based model for detecting vulnerabilities in C/C++ code. By leveraging attention weights, it supports both function-level and statement-level predictions, highlighting lines that are likely to be vulnerable.
- **CodeBERT** (Z. Feng et al., 2020) is a transformer-based model pretrained on diverse programming languages. To evaluate its transferability to security tasks, A model is fine-tuned on each dataset to classify vulnerable from non-vulnerable code.
- **GPT OSS Safeguard 120B** (OpenAI, 2025) is a 120B-parameter open-weight large language model developed by OpenAI, optimized for safety-oriented reasoning and high-accuracy predictions. We access the model via the Amazon Bedrock API, the most

Table 4
Summary of baseline models: architecture, features, and representation.

Model	Year	Architecture			Features		Source code	
		DL type	Classifier	Granularity	Code Rep.	Embedding	Type	Language
ASTGRU	2020	Bi-GRU	Softmax	Func.	AST	Word2Vec	AST graph	C/C++
IVDetect	2021	GCN	ReLU	Func.	AST/PDG	GloVe	Code	C/C++, Java
AMPLE	2023	GCN	Softmax	Func.	Code/CFG/DFG	Edge-aware	Code graph	C/C++
VulBG	2023	Linear	MLP	Func.	CFG/PDG/Code	CodeBERT	Code	C/C++
JLineVD	2024	GAT	MLP	Stat.	PDG/CDG	CodeBERT	Code	Java
SVDet	2024	Transformer	Linear	Stat.	Flat Code	JavaBERT	Code	Java
LineVul	2022	Transformer	Linear	Func.	Flat Code	CodeBERT	Code	C/C++
MLAF-VD	2025	GNN	FNN	Func.	CPG	CodeBERT	Code	C/C++
RoS-Dex	2026	Transformer	MLP	Func.	Flat Code	CodeT5+	Code	C/C++
CodeBert	2020	Transformer	Linear	Stat.	Flat Code	CodeBERT	Code	Any
GPT	2025	LLM	–	Func.	Code	–	Code	Any
ANGEL	2025	Graph/Transf.	MLP	Func.	Simplified CPG	Word2Vec	CPG	C/C++

current GPT-based model available under our license, and adapt it for source code vulnerability detection. Two evaluation settings are considered. First (*GPT*), we directly query the model with test samples and collect binary decisions (*Safe* or *Vulnerable*). Second (*GTP+*), we adopt a contextual empowering setup in which three code examples from the training set are included as contextual examples before querying the model on each test instance. Both settings employ a consistent message prompt template, illustrated below:

```
conversation = [
    {"role": "user",
    "content":[{
    "text":( "You are a software vulnerability analyst.\n"
      "Respond ONLY with 'Safe' or 'Vulnerable' depending on whether "
    "the code below contains any vulnerability.\n"
     "No explanations.\n\n"
    f"{code}")} ] }]
```

5. Results

5.1. RQ1: Can clustering-based approximation in the embedding space effectively model inter-function dependencies in project call graphs?

To validate the baseline edge approximations, we employed an existing static source code analysis tool for call graph construction, namely `Doxygen`,[1] to generate the dependency graphs. The process is carried out in several steps: (1) the source code of each project is organized into separate folders (e.g., QEMU and FFmpeg); (2) the Doxygen tool is executed on each folder to extract project documentation, including call graph information; (3) the resulting `.dot` files, each describing the local neighborhood of a function, were collected for every project; and (4) we use a Python script to merge all `.dot` files into a single project-level dependency graph by connecting function callers and callees based on their identifiers. This procedure yielded a function-level call graph with 14 794 nodes and 57 255 edges for QEMU, and 8284 nodes and 14 211 edges for FFmpeg, where the number of nodes corresponds to the number of functions in each project.

As an alternative, we also experimented with a token-based approach that produced a comparable number of edges. In this method, a token-level code search is employed to extract call graphs by parsing each project's source code, identifying function (or method) names, and searching for their occurrences across other functions of the same project. An edge is then added whenever a function name appears in the body of another function, thereby denoting an invocation relationship. In this representation, a directed edge between two functions explicitly captures a call relationship, identified solely on the basis of function names.

On the other hand, we construct the dependency graph from the embedding space and compare it with the static construction methods. Specifically, we start from graph-level representations obtained during feature generation using CodeBERT and standardize their dimensionality using GraphSAGE and Node2Vec (graph-to-vectors methods). GraphSAGE aggregates neighborhood information and produces fixed-size node embeddings, while Node2Vec captures structural similarity through biased random walks on the graph. The refined vector of each function, resized to a length of 128, is then clustered using DBSCAN. This density-based algorithm effectively identifies groups of semantically related functions without requiring a predefined number of clusters (k). We set the neighborhood radius to $\delta = 0.8$ and the minimum number of samples to $minPts = 2$. The choice of δ was empirically determined to optimize dependency recall. The *minPts* parameter is based on the idea that a function should be connected to at least one other function within the same project. An edge is established between two functions if they belong to the same DBSCAN cluster, resulting in an embedding-based dependency graph.

Using this approach, we generated alternative graphs as shown in Table 5 with 7255 nodes and 26 313 885 edges for FFmpeg (GraphSAGE) and 14 284 nodes and 102 009 186 edges for QEMU (GraphSAGE). Compared to the true call-graph representations (8284 nodes, 14 211 edges for FFmpeg; 14 794 nodes, 29 993 edges for QEMU), these embedding-based graphs show a drastic increase in connectivity and density.

While this embedding-based construction introduces fewer nodes (due to unparsed functions), it vastly increases the number of edges, preserving a high proportion of true call dependencies. For GraphSAGE, the recall rate reaches 82.4% on QEMU and 72.9% on FFmpeg, indicating that the majority of true call dependencies are retained. However, this comes at the cost of a high false connection rate: 99.95% for QEMU and 99.96% for FFmpeg, reflecting the dense nature of embedding-based similarity graphs. In contrast, Node2Vec achieves lower recall rates of 33.15% (QEMU) and 46.72% (FFmpeg), while introducing fewer spurious edges, highlighting a trade-off between dependency coverage and graph noise.

A preserved/recall connection is defined as a directed edge present in both the baseline call graph and the clustering-based graph, where the correspondence between functions is established using either function names or node identifiers (i, j). This indicates that GraphSAGE provides a much stronger approximation, though at the cost of introducing noisy connections. Moreover, the clustering uncovers higher-level similarities between functions that are not captured by direct inter-function dependencies alone, highlighting code behavior in the embedding space. However, we argue that part of the large number of noisy edges may arise from the directed nature of the graph, in which an edge (i, j) is treated as distinct from (j, i). This representation inherently increases edge counts compared to an undirected view and can amplify spurious connections introduced by the clustering process.

[1] https://www.doxygen.nl/index.html.

Table 5
Edge preservation between baseline and embedding-based clustering methods.

Project	Method (M)	B nodes	B edges	M nodes	M edges	Preserved edges	Preservation (%)	Time (s)
QEMU	GraphSAGE	14 794	57 255	14 284	102 009.86	47 317	82.4	130.81
QEMU	Node2Vec	14 794	57 255	14 284	25 502 299	18 982	33.15	1562.25
FFmpeg	GraphSAGE	8284	14 211	7255	26 313 885	10 364	72.93	89.95
FFmpeg	Node2Vec	8284	14 211	7255	6 578 472	3871	46.72	1302.48

RQ1: Clustering in the learned embedding space can approximate inter-function dependencies, preserving up to **82.4%** of true call relationships (GraphSAGE on QEMU) while capturing additional latent behavioral links. Although it introduces noise, this method offers a scalable and viable alternative to static or dynamic call graph construction, particularly when full program analysis is infeasible.

5.2. RQ2: How does incorporating inter-function dependencies influence the performance of GNN-based models for function and statement-level vulnerability detection?

To address this research question, we evaluated three model configurations across both *FFmpeg+QEMU* and *ProjectKB* datasets for each code embedding technique (i.e., Word2Vec, SBERT, and CodeBERT):

We configure the model variant as (i) a **baseline** model without inter-function dependency integration; (ii) a model with **concatenation-based** integration; and (iii) a model with **attention-based** fusion. For graph-to-vector transformation in the dependency construction, we adopted the **DBSCAN** + **GraphSAGE** framework, as it requires fewer computational resources than the random walk process when using Node2Vec. Tables 6 and 7 summarize the results in terms of Precision, Recall, and F1-score at both function and statement levels.

Across both datasets, integrating inter-function dependencies yielded consistent improvements. For example, on the FFmpeg+QEMU dataset with *CodeBERT*, recall improved from 0.67 (baseline) to 0.83 with concatenation and 0.82 with attention. With *Word2Vec*, recall increased from 0.50 to 0.61 with both methods, while *Sentence-BERT* feature embedding shows smaller but still positive improvements, with recall rising from 0.51 to 0.53 using concatenation.

In the ProjectKB dataset, at the function level, CodeBERT's F1 increased from 0.66 to 0.76 (+0.10) using concatenation, while attention produced a smaller improvement. Similarly, at the statement level, it improved the F1-score from 0.48 to 0.52 (+0.04) and 0.49 (+0.01), respectively, with concatenation and attention.

Using a different code embedding, SBERT improved substantially: from 0.42 (baseline) to 0.53 (+0.11) with concatenation and to 0.55 (+0.13) with attention at the statement level, while performing poorly at the function level.

In addition to absolute gains, we observed reduced metric variance (reported in Tables 6 and 7) when inter-function dependency features were integrated. In several configurations (e.g., Word2Vec and SBERT), we observe reductions in the standard deviation of metrics (e.g., F1 $\pm 2\%$ to $\pm 1\%$), which corresponds to variance decreasing by up to a factor of 4 and is more noticeable at the statement level. To assess whether the observed improvements are statistically meaningful, we conducted paired t-tests between the paired elements of the baseline and dependency-setting models. The results in Table 8 show that statistically significant differences ($p < 0.05$, marked in bold) occur in several settings, most notably for the FFmpeg+QEMU dataset and for the CodeBERT and Word2Vec embeddings. Whereas many of the improvements on ProjectKB, especially at the statement level, do not reach significance. We hypothesize that this discrepancy arises because FFmpeg+QEMU contains procedural code written in the C programming language that comprises more interconnected functions, in which dependency information adds substantial contextual value. Instead, ProjectKB contains Java code in which functions are more self-contained, the code is organized by classes, and even because functions are extracted from various open-source projects, making the dependencies edges potentially noisy.

These results confirm that modeling inter-function dependencies allows the system to capture vulnerability propagation beyond isolated functions. Although both fusion mechanisms yield consistent improvements, concatenation slightly outperforms attention on average. We attribute this to the fact that concatenation explicitly expands the feature space by stacking node-level and dependency-level representations, thereby increasing model capacity and preserving complementary information. In contrast, attention reweights features within the same dimensional space. Attention, however, provides more balanced representations and demonstrates competitive performance in settings where the feature vector is less representative.

RQ2: Inter-function dependency modeling, implemented with GraphSAGE, significantly improves vulnerability detection across datasets, with relative recall gains up to +**16%** for FFmpeg+QEMU and +**13%** for ProjectKB. Integration also reduces variance, increasing training stability. On average, concatenation offers slightly better improvements than attention, though both are effective in leveraging inter-function context.

5.3. RQ3: To what extent do different code embedding techniques (e.g., CodeBERT, SentenceBERT, Word2Vec) impact the node representations in vulnerability detection tasks?

We in-depth analyzed the collected results reported on Tables 6 and 7 to examine the influence of embedding choice on vulnerability detection performance (see Section 3.2.2).

CodeBERT consistently outperformed other embeddings across datasets and integration strategies. For instance, on FFmpeg+QEMU, CodeBERT with concatenation achieved a function-level F1-score of 0.82, compared to 0.51 for SBERT, and 0.61 for Word2Vec. The relative improvement in F1-score from the baseline was +0.16 for CodeBERT, +0.02 for SBERT, and +0.11 for Word2Vec. On ProjectKB at the statement level, SBERT achieved its largest F1's gain with respect to the baseline (+0.13) with attention, surpassing the gain obtained with concatenation (+0.11). We note that no single embedding method achieved top performance at both statement and function levels. CodeBERT (768 dimensions) excelled at function-level tasks, likely because its large contextual capacity accurately captures broader semantic and structural cues of entire functions. However, for statement-level tasks, its fixed-length vectors may suffer from excessive padding when representing short code lines, diluting fine-grained information. In contrast, SBERT (384 dimensions) and Word2Vec (100 dimensions) may be better suited for shorter sequences but lack the global context modeling power of CodeBERT.

We evaluated the impact of different graph-to-vector embeddings and clustering techniques (see Section 3.2.3), focusing on *Node2Vec* and *GraphSAGE* for embeddings, and *k-means* ($k = 2$) and *DBSCAN* for clustering. The tested configurations included *k-means* + *Node2Vec*, *k-means* + *GraphSAGE*, and *DBSCAN* + *Node2Vec*. Results showed only marginal differences (Table 9). While *k-means* + *Node2Vec* and *k-means* + *GraphSAGE* performed similarly, *DBSCAN* remains appealing since it does not require a predefined number of clusters. However, *Node2Vec* proved computationally costly (see Table 5), reduced edge coverage compared to the baseline, and offered no improvements over

Table 6
Models' performance with and without inter-function dependencies integration on FFmpeg+QEMU. Each metric is followed by a standard deviation σ: ($m \pm \sigma\%$).

Embed.	No dependency integration			Concatenate			Attention		
	Prec.	F1	Rec.	Prec.	F1	Rec.	Prec.	F1	Rec.
CodeBERT	0.68 ± 2	0.66 ± 1	0.67 ± 4	0.82 ± 2	0.82 ± 1	0.83 ± 5	0.81 ± 2	0.81 ± 1	0.82 ± 2
SBERT	0.47 ± 5	0.49 ± 6	0.51 ± 6	0.47 ± 3	0.51 ± 2	0.53 ± 3	0.49 ± 2	0.42 ± 3	0.50 ± 1
Word2Vec	0.50 ± 1	0.50 ± 2	0.50 ± 3	0.60 ± 5	0.61 ± 4	0.61 ± 2	0.60 ± 2	0.61 ± 1	0.61 ± 2

Table 7
Models' performance with and without inter-function dependencies integration on ProjectKB. Each metric is followed by a standard deviation σ: ($m \pm \sigma\%$).

Embed.	No dependency integration			Concatenate			Attention		
	Prec.	F1	Rec.	Prec.	F1	Rec.	Prec.	F1	Rec.
Function-level									
CodeBERT	0.65 ± 12	0.66 ± 10	0.87 ± 5	0.71 ± 4	0.76 ± 3	0.92 ± 2	0.66 ± 4	0.68 ± 5	0.88 ± 3
SBERT	0.55 ± 18	0.48 ± 15	0.52 ± 10	0.62 ± 1.5	0.56 ± 1	0.54 ± 0.8	0.58 ± 5	0.54 ± 2	0.56 ± 1.5
Word2Vec	0.45 ± 18	0.47 ± 4	0.49 ± 3	0.49 ± 8	0.49 ± 0.4	0.50 ± 1.2	0.50 ± 6	0.50 ± 10	0.51 ± 2
Statement-level									
CodeBERT	0.47 ± 22	0.48 ± 13	0.51 ± 15	0.52 ± 5	0.52 ± 4	0.51 ± 3	0.49 ± 8	0.49 ± 6	0.50 ± 5
SBERT	0.40 ± 25	0.42 ± 20	0.50 ± 10	0.57 ± 2	0.53 ± 1.5	0.51 ± 1	0.57 ± 4	0.55 ± 3	0.54 ± 2
Word2Vec	0.50 ± 12	0.46 ± 16	0.53 ± 10	0.51 ± 2	0.47 ± 2	0.54 ± 1.8	0.51 ± 12	0.47 ± 15	0.54 ± 9

Table 8
t-statistics of the baseline model against dependency construction version.

Embed.	Metric	Baseline vs. Concatenation			Baseline vs. Attention		
		Prec	F1	Rec	Prec	F1	Rec
FFmpeg+QEMU (Function-level)							
CodeBERT	*t*-stat	−31.49	−62.59	−12.29	−27.26	−78.29	−15.15
	p-value	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**
SBERT	*t*-stat	−0.40	−0.61	−0.83	−1.19	7.73	1.33
	p-value	0.6890	0.5451	0.4147	0.2451	**<0.0001**	0.1948
Word2Vec	*t*-stat	−12.60	−12.24	−22.19	−27.95	−28.21	−23.39
	p-value	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**	**<0.0001**
ProjectKB (Function-level)							
CodeBERT	*t*-stat	−2.02	−4.35	−4.40	−0.34	0.24	0.56
	p-value	0.0525	**0.0002**	**0.0001**	0.7328	0.8132	0.5807
SBERT	*t*-stat	−1.46	−4.44	−1.60	−0.20	−3.39	−2.72
	p-value	0.1556	**0.0001**	0.1196	0.8447	**0.0020**	**0.0110**
Word2Vec	*t*-stat	−2.57	−1.51	−1.98	−2.98	−1.70	−3.76
	p-value	**0.0154**	0.1420	0.0567	**0.0058**	0.0999	**0.0008**
ProjectKB (Statement-level)							
CodeBERT	*t*-stat	0.23	−1.59	−1.27	0.79	0.79	−0.84
	p-value	0.8235	0.1232	0.2156	0.4350	0.4355	0.4101
SBERT	*t*-stat	−3.91	−3.45	−0.95	−4.01	−4.06	−2.58
	p-value	**0.0005**	**0.0017**	0.3483	**0.0004**	**0.0003**	**0.0153**
Word2Vec	*t*-stat	−0.40	−1.37	−0.95	−0.16	−1.91	−1.51
	p-value	0.6929	0.1811	0.3492	0.8771	0.0664	0.1425

GraphSAGE, which achieved an approximate twelvefold speedup. The dependency graphs constructed using Node2Vec also introduce a substantially higher number of edges compared to the baseline graphs and fewer than GraphSAGE. While this increases connectivity, it also leads to denser neighborhoods. The results of Table 9 indicate a metric difference. However, this difference is not directly associated with noisy additional edges but instead with the general unicity of the constructed graph.

To further assess the impact of feature integration methodologies, we conducted experiments on the Big-Vul dataset using CodeBERT embeddings and GraphSAGE in the inter-dependency construction, examining both imbalanced and balanced training data distributions. In this context, the balanced data consists of a random downsample of the most represented class to match the size of the less represented class.

Table 10 summarizes the results. While performance varies across data distributions, both concatenation and attention outperform the base model. Notably, feature fusion improves recall in both settings, indicating more effective propagation of vulnerability-related signals, with attention achieving the highest function-level F1-score under balanced training and concatenation showing stronger robustness under imbalance.

Table 9
Graph-to-vector conversion methods for GL performance comparison on ProjectKB and FFmpeg+QEMU datasets at function level.

Method	FFmpeg+QEMU			ProjectKB		
	Pre	F1	Recall	Pre	F1	Recall
k-means + Node2Vec	0.79	0.79	80	0.70	0.77	0.89
k-means + GraphSAGE	0.82	0.82	0.82	0.71	0.78	0.89
DBSCAN + Node2Vec	0.81	0.80	0.80	0.69	0.77	0.88

Table 10
The performance of SCVdet on the Big-Vul dataset, utilizing CodeBERT embeddings, is evaluated under both balanced and imbalanced settings. The results presented are averaged over three runs.

Model	Big-Vul	Statement level			Function level		
		Pre	F1	Rec	Pre	F1	Rec
Base model	Imbalanced	0.497	0.498	0.500	0.671	0.692	0.722
Base model	Balanced	0.513	0.502	0.525	0.683	0.665	0.873
Concatenate	Imbalanced	0.535	0.556	0.581	0.687	0.760	0.838
Concatenate	Balanced	0.564	0.582	0.601	0.684	0.778	0.928
Attention	Imbalanced	0.517	0.551	0.567	0.701	0.766	0.851
Attention	Balanced	0.535	0.565	0.595	0.691	0.799	0.902

RQ3: Embedding choice significantly impacts vulnerability detection, with CodeBERT providing the best function-level performance (+**16%** F1-score gain) and SBERT achieving the largest relative improvement (+**13%**) at the statement level. This variation is partly explained by embedding dimensionality—high-dimensional embeddings excel at capturing global function semantics, while lower-dimensional ones may better preserve fine-grained statement-level detail. Effective integration of embeddings with graph-derived context is key to maximizing detection accuracy. Additionally, the dataset structure affects model performance, as observed in the performance improvement on the big-vul dataset. These results highlight a trade-off between robustness to imbalance (concatenation) and refined feature selection (attention).

5.4. RQ4: How does SCVdet improve source code vulnerability detection compared to its counterparts?

Table 11 presents a comprehensive comparison between **SCVdet** and the latest sequence-based, graph-based, and LLM-based vulnerability detection approaches in four benchmark datasets: FFmpeg+QEMU, ProjectKB, Big-Vul, and CVEFixes. We report results at both the function and statement levels whenever supported by the compared models. Here, "–" denotes cases where a given prediction granularity (i.e., statement level) is not applicable for a model, typically because the method is inherently designed to operate only at the function level.

Function-level results on FFmpeg+QEMU: SCVdet achieves the best overall performance among all evaluated approaches, with a precision of 0.826, recall of 0.836, and F1-score of 0.829. Compared to the baseline, VulBG (F1 = 0.561), SCVdet improves the F1-score by +0.268 and precision by +0.308. It also substantially outperforms recent learning-based approaches such as ASTGRU, LineVul, MLAF-VD, RoS-Dex, and both GPT-based versions used. These gains demonstrate that explicitly modeling inter-function dependencies via our clustering-based dependency graph leads to more reliable vulnerability detection than purely sequential, structural, or local graph representations. We argue that this performance is mainly due to the fact that many code functions in the FFmpeg+QEMU dataset are written in procedural C and share relative connections (e.g., see Fig. 1(a)), which are largely not captured by other models.

Function-level results on ProjectKB: SCVdet remains competitive with the latest methods while showing the highest recall (0.924). Although CodeBert and SVDet achieve the highest precision (0.874 and 0.86, respectively), SCVdet outperforms IVDetect by +0.10 in precision and +0.045 in F1-score. Compared to JLineVD, SCVdet improves recall by +0.024. The other pre-trained models, both GPT versions, achieved lower precision and recall, underperforming with respect models such as LineVul and IVDetect. We observe that SCVdet underperforms BERT-like methods (i.e., CodeBert and SVDet) in terms of precision and F1-score. We argue that this is mainly due to the fact that in the Java code of the ProjectKB dataset, functions are more self-contained thanks to the class-level organization of the code and the fact that such functions are extracted from different projects. However, this also highlights the need to improve the construction of the dependency graph by better considering the code structure of Object-Oriented programming languages. The introduction of noisy edges described in Section 5.1 may have contributed to this decline in performance.

Statement-level results on ProjectKB: At the statement level, SCVdet achieves a balanced performance with a precision of 0.576, recall of 0.545, and F1-score of 0.553. It slightly outperforms SVDet and JLineVD in F1-score by +0.153 and +0.004, respectively, while closely matching the precision of JLineVD (difference of −0.01). This confirms that SCVdet's dependency-aware representations can remain effective even when predictions are derived from fine-grained statement-level evidence.

Results on Big-Vul: SCVdet achieves the highest recall (0.941) among all evaluated methods, surpassing strong baselines such as JLineVD (0.930) and IVDetect (0.720). While its precision (0.724) is lower than LineVul and MLAF-VD, SCVdet provides a better recall–precision balance than most learning-based competitors, leading to a competitive F1-score of 0.788. At the statement level, SCVdet further improves recall to 0.697, outperforming JLineVD (i.e., an improved version of LineVD), which indicates its robustness in identifying vulnerable code lines. The dominance of SCVdet's recall underscores its capability to address the limitations of existing models in capturing long-range dependencies through the use of a constructed dependency graph. However, its lower precision may stem from the use of a similarity-based method to construct it. This method could lead to interconnected functions that, while exhibiting feature-level connections, do not necessarily share dependency knowledge. Another aspect negatively impacting the SCVdet's performance is the presence of object-oriented code in the dataset (about 20% of BigVul code is estimated to be from C++ projects), in which the class-oriented organization reduces the capability of our tool in identifying cross-function dependencies.

Results on CVEFixes: SCVdet consistently improves recall over GPT-based and LineVul baselines, achieving 0.861 at the function level and 0.582 at the statement level. While CodeBERT attains higher precision at the function level, SCVdet provides a more balanced trade-off, yielding an F1-score improvement over GPT, GPT+, and LineVul. Compared to JLineVD, SCVdet demonstrates higher recall at both granularities, reinforcing its suitability for vulnerability discovery tasks where coverage is critical. Furthermore, we can argue that the lower precision observed, with respect to the one measured in other datasets, is mainly due to the fact that Python is a strongly dynamic language, in which types and function calls are known, in particular, at runtime. This types of function interconnection is not adequately captured from the similarity-based method in this work.

These results collectively demonstrate that SCVdet offers several practical advantages:

- **Dependency-aware learning:** By explicitly incorporating inter-function dependencies, SCVdet captures both direct and latent vulnerability propagation patterns that are overlooked by token-based, sequence-only, or purely intra-function graph approaches. Its consistently higher recall compared to models such as VulBG, IVDetect, and ASTGRU confirms its superior vulnerability coverage.
- **Multi-granularity robustness:** SCVdet generalizes effectively across function- and statement-level predictions with limited performance degradation, unlike many baselines that are restricted to a single analysis granularity.

Table 11
Comparison of model performance with state-of-the-art approaches at function and statement levels.

Dataset	Model	Function-level			Statement-level		
		Prec.	F1	Rec.	Prec.	F1	Rec.
FFmpeg+ QEMU	GPT	0.555	0.486	0.531	–	–	–
	GPT+	0.571	0.564	0.56	–	–	–
	AMPLE	0.556	0.669	0.839	–	–	–
	ASTGRU	0.538	0.518	0.499	–	–	–
	VulBG	0.518	0.561	0.612	–	–	–
	IVDetect	0.600	0.650	0.720	–	–	–
	LineVul	0.663	0.601	0.549	–	–	–
	MLAF-VD	0.635	0.668	0.706	–	–	–
	RoS-Dex	0.583	0.684	0.819	–	–	–
	CodeBert	0.618	0.544	0.486	–	–	–
	SCVdet	**0.826**	**0.829**	**0.836**	–	–	–
Project KB	GPT	0.572	0.538	0.534	–	–	–
	GPT+	0.594	0.602	0.611	–	–	–
	IVDetect	0.613	0.713	0.852	–	–	–
	LineVul	0.859	0.746	0.659	–	–	–
	CodeBert	**0.874**	0.810	0.782	0.35	0.284	0.241
	SVDet	0.860	**0.881**	0.900	0.433	0.402	**0.545**
	JLineVD	0.714	0.760	0.812	**0.586**	0.549	0.518
	SCVdet	0.713	0.758	**0.924**	0.576	**0.553**	**0.545**
Big-Vul	AMPLE	0.165	0.238	0.430	–	–	–
	IVDetect	0.230	0.350	0.720	–	–	–
	LineVul	**0.941**	**0.920**	0.901	–	–	–
	MLAF-VD	0.933	0.884	0.840	–	–	–
	RoS-Dex	0.710	0.613	0.539	–	–	–
	ANGEL	0.270	0.315	0.378	–	–	–
	JLineVD	0.816	0.863	0.930	**0.586**	0.530	0.520
	SCVdet	0.724	0.788	**0.941**	0.558	**0.587**	**0.697**
CVEFixes	GPT	0.554	0.539	0.535	–	–	–
	GPT+	0.575	0.568	0.564	–	–	–
	LineVul	0.548	0.535	0.522	–	–	–
	CodeBert	**0.892**	**0.826**	0.783	0.487	0.493	0.501
	JLineVD	0.687	0.696	0.707	**0.716**	0.570	0.542
	SCVdet	0.670	0.752	**0.861**	0.661	**0.619**	**0.582**

- **Imbalance-resilient detection:** Across highly imbalanced datasets such as ProjectKB and Big-Vul, SCVdet maintains strong recall and competitive F1-scores, making it particularly well suited for real-world vulnerability auditing scenarios.

RQ4: Overall, the results confirm that SCVdet not only competes with but frequently surpasses state-of-the-art vulnerability detection approaches across multiple datasets and granularities. Its consistent recall gains, balanced F1-scores, and dependency-aware design make it a strong and practical solution for large-scale source code vulnerability detection.

6. Further analysis and discussion

To further assess the effectiveness of our proposed **SCVdet** tool, which explicitly models and integrates a dependency graph between functions, we perform two complementary analyses based on a detailed inspection of confusion matrices to better understand model prediction behavior.

Fig. 4 presents the confusion matrices for the best SCVdet configurations across embedding strategies (i.e., CodeBert, SBert, and Word2vec), when the dependency graph is constructed with DBSCAN + GraphSAGE. On both datasets, models using *CodeBERT* embeddings (features in graph) achieved the highest **true-positive rate (TPR)** for vulnerable functions. Importantly, the confusion matrix analysis provides a fine-grained and interpretable view of class predictions, which is particularly useful for imbalanced datasets such as ProjectKB, where aggregate metrics can obscure poor minority-class behavior or overestimate overall performance.

In particular, on ProjectKB, SCVdet correctly identified *all* vulnerable functions in the test set when using direct function-level predictions derived from function-level embeddings. While this result demonstrates excellent recall, it also introduces a limited number of false positives, i.e., some non-vulnerable functions misclassified as vulnerable (first confusion matrix), reflecting a recall-oriented trade-off that is clearly exposed by the confusion matrix representation rather than by summary metrics alone.

We employed the node/statement to function aggregation (i.e., identification of the vulnerable functions derived from the identified vulnerable statements or node in the graph-based code representation) strategy exclusively on ProjectKB since the considered methods and tools for C/C++ work only at function-level. Interestingly, when deriving function-level predictions from statement features, *Word2Vec* embeddings exhibited a **TPR improvement (see** Fig. 4 **position (3,2))** compared to the direct function-level prediction approach. This observation implies that simpler embedding models can benefit more from fine-grained, statement-level aggregation, where localized vulnerability signals are explicitly captured. Conversely, the observed decrease in TPR for CodeBERT and SBERT under the same aggregation strategy may indicate a *semantic dilution effect*, whereby aggregating node-level representations weakens the higher-level semantic coherence encoded in holistic function embeddings. We further analyze the learned feature space at different stages of the model, including the raw input features, the representations produced by the GAT layers, and the final MLP outputs.

t-SNE is a technique used to reduce complex data to a low-dimensional space (i.e., 2 or 3 dimensions), thus making them visible in a 2/3d plot. t-SNE can be used to visually identify hidden clusters and patterns by observing similar data points close together in the low-dimensional space. The t-SNE analysis shown in Fig. 5 illustrates feature distributions obtained when training the model on the imbalanced Big-Vul dataset, with dependency information incorporated via the concatenation fusion mechanism (metrics in Table 10).

In the first row of Fig. 5, the first two subfigures visualize the raw input features at the function and node (statement) levels, respectively. While function-level features exhibit a structured distribution in the embedding space, node-level features remain largely intermixed, indicating limited intrinsic separability. The last two subfigures in the same row show the corresponding output features of a model trained without dependency information, where the representations are partially reorganized but still exhibit substantial class overlap.

The second row presents the representations learned at different GAT layers, followed by the dependency-enhanced features. Although the visual differences are subtle, a shift in feature distribution can be observed when dependency information is incorporated. The inter-function dependencies contribute additional contextual signals that refine the learned representations. In particular, vulnerable node features that were initially scattered among non-vulnerable samples (e.g., in the top-left subfigure) become more concentrated in specific regions of the embedding space, albeit with some remaining overlap. A similar trend is observed at the function level: comparing subfigures (1,4) and (2,4), dependency-enhanced features show a slight reduction in the overlap between vulnerable and non-vulnerable functions. These observations indicate that dependency-aware feature integration improves the model's ability to capture structural relationships and vulnerability-related patterns across both node and function levels.

6.1. Threats to validity

Internal. Our approach operates on datasets in which each sample corresponds to a single function, often without explicit access to broader project-level context. This limitation can introduce noise when constructing inter-function dependency graphs, as many relationships must

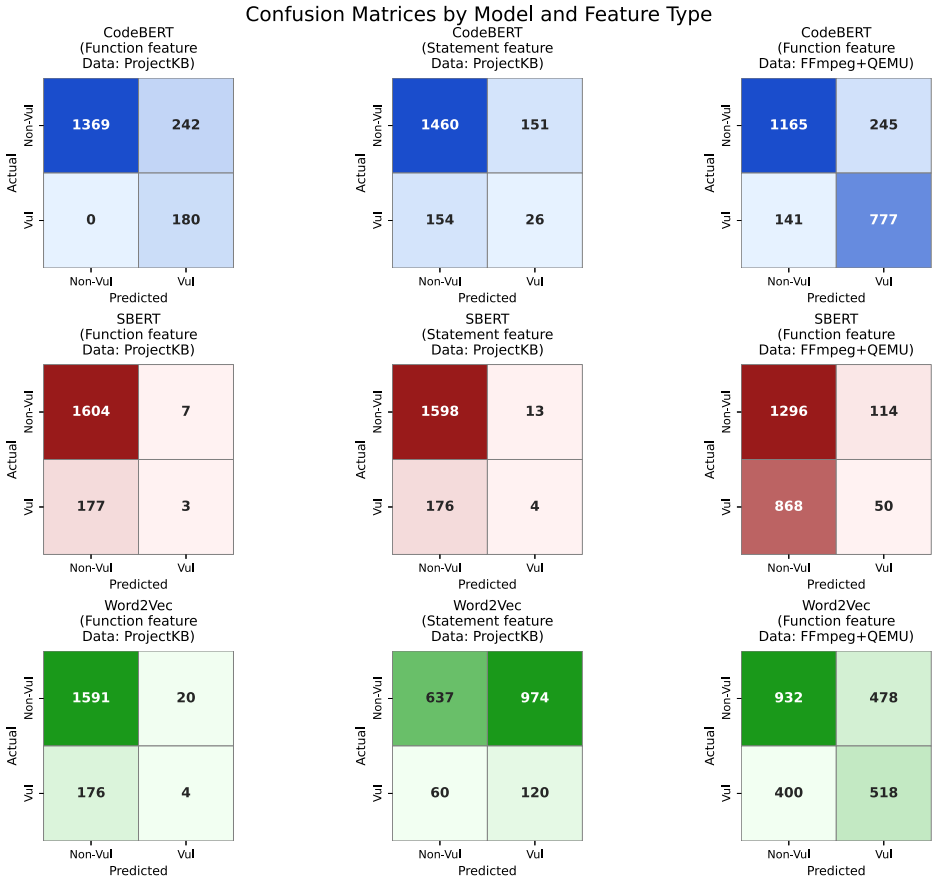


Fig. 4. Confusion matrices for the best-performing SCVdet configurations, highlighting differences in correct and incorrect classifications across embedding strategies. The first row displays function-level results using CodeBERT for feature generation, while the second and third rows present SBERT and Word2vec, respectively. The first two columns represent the ProjectKb dataset under different feature settings, while the third column shows the FFmpeg+QEMU dataset.

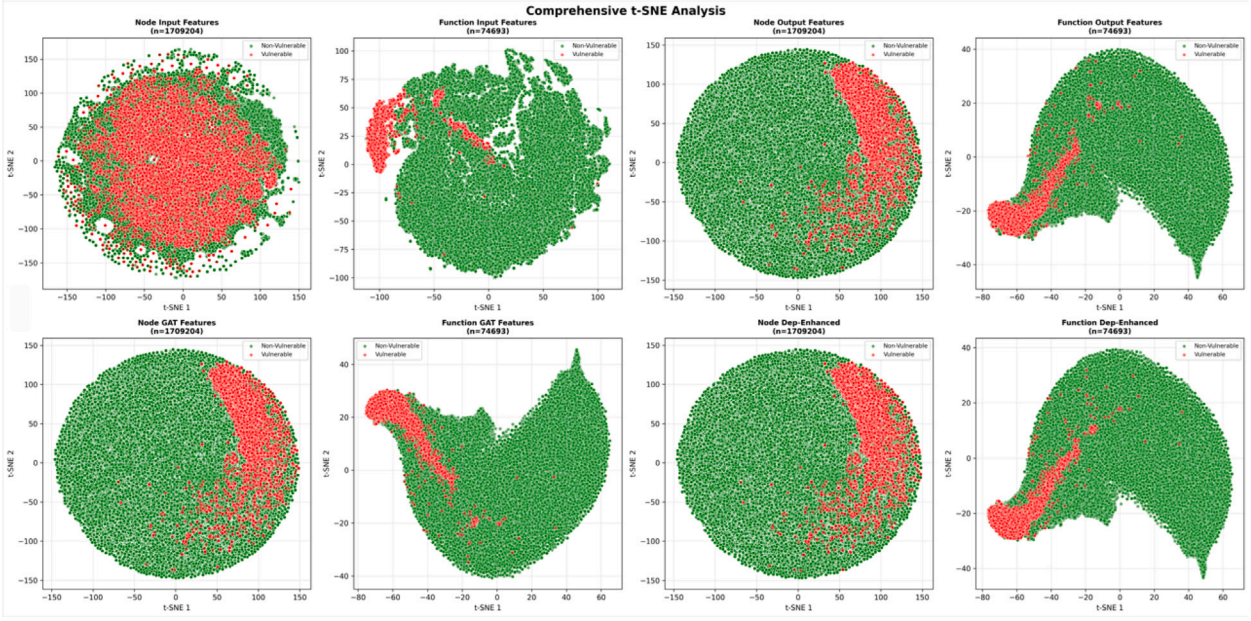


Fig. 5. t-SNE visualization of feature representations learned on the imbalanced Big-Vul dataset. Red points denote vulnerable samples, while green points denote non-vulnerable ones. The figure compares raw input features, GAT-layer representations, and final MLP outputs at both node (statement) and function levels, with and without dependency-enhanced feature fusion via concatenation. Here, n denotes the number of samples used during training and evaluation.

be inferred rather than directly observed. In addition, vulnerability labels are primarily provided at the function level, which may reduce the reliability of fine-grained (statement-level) predictions when vulnerabilities affect only specific code regions. While *ProjectKB, Big-Vul, and CVEFixes* contain both vulnerable functions before and after fixes, enabling partial statement-level supervision, *FFmpeg+QEMU* lacks such annotations, restricting supervision to the function level. Furthermore, a limited subset of samples is discarded when the source code cannot be parsed by *Joern*, typically due to syntactic or structural inconsistencies. These samples are excluded to ensure that graphs and representations are constructed only from well-formed code, thereby avoiding the introduction of noise from erroneous samples.

External. The generalizability of our findings may be limited by the scope of the evaluated datasets and programming languages. Our experiments focus on open-source projects written in C/C++, Java, and Python, and performance on other languages (e.g., JavaScript or PHP) or proprietary codebases with different architectural patterns remains unexplored. Moreover, since our approach relies on code CPGs generated by *Joern*, parser limitations may affect portability across analysis tools or environments. Although recent versions of *Joern* offer improved multi-language support, we focused on a subset of its capabilities due to the high computational cost of full graph extraction. As a consequence, some functions that could not be parsed were excluded. This resulted in a final dataset of 74 693 training samples, 10 086 validation samples, and 9468 test samples, organized under an 80:10:10 split, from the relatively larger dataset, such as Big-Vul.

Statistical. The evaluation of multiple model variants introduces potential variance stemming from random initialization and pronounced class imbalance, particularly in *ProjectKB*. Consequently, small performance differences may not always reflect statistically significant improvements. Future work will incorporate formal statistical significance testing and confidence interval estimation to strengthen the robustness of comparative results.

Computational constraints and clustering motivation. Constructing complete inter-function dependency graphs for large-scale codebases is computationally expensive in both time and memory. In our experiments, generating CPGs for datasets such as *FFmpeg+QEMU*, *ProjectKB, CVEFixes, and Big-Vul* required several days, while feature graph generation for larger datasets such as *Big-Vul* took up to six days on our local infrastructure. Model training time also scales with dataset size: training SCVdet on the full Big-Vul dataset required approximately five hours, whereas significantly shorter runtimes were observed for smaller datasets (i.e, up to a maximum of 3 h).

To address the prohibitive cost of exhaustive dependency construction (Yamaguchi et al., 2014; Y. Li et al., 2021), we employ clustering-based dependency approximation. Specifically, functions are embedded and grouped using density-based algorithms such as DBSCAN to identify semantically and structurally similar neighborhoods. This strategy significantly reduces the reliance on full static dependency extraction while preserving meaningful inter-function relationships. Although this approximation may omit certain fine-grained dependencies, it offers a practical trade-off between scalability and representational fidelity, particularly in settings where complete call graphs are unavailable or infeasible to compute. Future work will explore hybrid approaches that combine static analysis with learned dependency inference to further improve accuracy without sacrificing scalability.

6.2. Interpretability and language generalization

SCVdet is language-agnostic and has been evaluated on both Java and C/C++ datasets. Adapting it to a new language requires only an update to the feature extraction pipeline (e.g., changing file path configurations from "`path.java`" to "`path.py`" or "`path.c`").

The model supports *multi-task learning*, predicting vulnerabilities at both function and statement (node) levels. For datasets with paired vulnerable and fixed function versions, such as *ProjectKB*, this enables line-level interpretability, similar to *LineVD* and *LineVul*.

Nonetheless, the model remains largely a black box. Future work will explore feature attribution methods (e.g., SHAP, integrated gradients) to highlight features most influencing each prediction. Such techniques could enhance transparency, guide developer remediation efforts, and enable even finer-grained predictions, potentially down to the token level.

7. Related work

Automated SVD has significantly advanced in recent years due to the emergence of novel AI-driven techniques, particularly sequential and graph-based methods. These approaches have demonstrated their potential as alternatives to traditional static analyzers, which are often plagued by high false positive rates in source code analysis (Sonnekalb et al., 2022; Shiri Harzevili et al., 2024; Yang and Li, 2024).

Several works have explored sequential and graph-based models for vulnerability detection. For instance, Hin et al. (2022) proposed LineVD, Fu and Tantithamthavorn (2022) introduced LineVul, Tran et al. (2025) developed DetectVul, and Ni et al. (2023) presented MVulD, among others. Although LineVD, LineVul, and DetectVul perform vulnerability detection at the statement level, they each rely on a single programming language; typically C/C++ or Python for DetectVul only, raising concerns about cross-language generalization. Moreover, the underlying modeling choices impose structural limitations: LineVul is a sequence-based model and operates on tokens, which hinders its ability to capture higher-level syntactic and semantic structure. DetectVul, although graph-based, extracts only a Python AST representation; this narrow design restricts applicability to other languages and yields a limited structural view of the code, omitting richer representations such as control-flow and data-flow.

Additionally, Haque et al. (2025) used class-weighted cross-entropy to address the fundamental imbalanced nature of the dataset, which consists of C/C++ source code, thus improving the true positive rate over 30 independent runs. While these methods have contributed to improving vulnerability detection, they generally analyze source code in isolation, without considering the interdependencies between functions. Furthermore, most existing techniques focus on vulnerability detection at the function level, predominantly for C/C++, with limited support for statement-level analysis.

To address these limitations, Li et al. (2024) introduced the first inter-procedural vulnerability dataset for C/C++, demonstrating that modeling function interactions improves function-level detection performance over traditional intra-procedural approaches. Similarly, Song et al. (2023) proposed HGIVul, a hypergraph convolution-based method that reconstructs inter-procedural control-flow graphs (ICFGs) to capture complex cross-function relationships. Their results outperform prior models such as VulDeePecker (Li et al., 2018), SySeVR (Z. Li et al., 2021), and DeepWukong (Cheng et al., 2021), highlighting the importance of dependency-aware analysis. However, despite their effectiveness, these approaches remain computationally expensive and limited to C/C++ code, restricting their practical applicability in real-world, multi-language environments. Moreover, these outperformed models are primarily sequence-based. Since they require limited code parsing before training on source code, this may cause the tools to learn from incorrectly structured code. Recent studies have attempted to capture vulnerabilities across multiple functions. Zhang et al. (2025) and Wu et al. (2023) explored vulnerability detection through vulnerability-specific inter-procedural slicing, learning program semantics to improve detection accuracy. However, their methods were limited to Java Web vulnerabilities and six C/C++ vulnerability types, and the static slicing technique led to high false positive rates. Moreover, since their approach relies on static analysis, it cannot effectively analyze vulnerabilities that require modeling of runtime

behavior. Subsequently, Yuan et al. (2023) proposed VulBG. This behavioral graph model extracts code snippets and constructs behavioral graphs using hierarchical clustering with a k-means algorithm to define graph neighborhoods.

Although VulBG demonstrates promising results in function-level vulnerability detection, several limitations constrain its broader applicability. First, the model lacks support for statement-level vulnerability identification, which reduces its ability to capture fine-grained issues within source code. Second, its reliance on data from a single programming language limits generalization across diverse software ecosystems. In addition, the code slicing process significantly reduces the fidelity of source code representation, often capturing only partial vulnerability information. Collectively, these factors lead to incomplete semantic coverage, preventing VulBG from fully modeling the complex behaviors that underlie real-world vulnerabilities. Additionally, Wang et al. (2025) demonstrates that modifications to code structure, including external function calls, can alter the predictions of both scratch-trained and pre-trained deep learning models for vulnerability detection.

Unlike these prior approaches, our work aims to construct an inter-function relationship graph that explicitly captures dependencies between functions to enhance vulnerability detection at the statement level. Our approach dynamically updates the function-level dependency graph, enabling the model to learn and evaluate the propagation of potential vulnerabilities across interconnected functions. By integrating both local (statement-level) and global (function-level) information via an attention and concatenation process, our method seeks to provide a more comprehensive and adaptable solution for vulnerability detection across diverse programming languages and code structures.

8. Conclusion

We presented **SCVdet**, a GAT-based model for source code vulnerability detection that leverages a clustering-based technique to approximate inter-function connections within a project. These approximated dependencies are integrated into the model through two fusion strategies, concatenation and attention, which enrich the learned representations with both local and global contextual information. This integration has led to consistent improvements in key performance metrics on two real-world datasets, demonstrating the effectiveness of capturing higher-level code dependencies for more accurate vulnerability detection. Beyond its performance gains, SCVdet offers a scalable alternative to full dependency graph construction, significantly reducing computational costs while retaining meaningful relational signals.

Future research will explore more precise and actionable methods for constructing dependency graphs, particularly regarding object-oriented and dynamic languages. This includes the investigation of hybrid approaches that combine static and dynamic dependencies. Additionally, we will extend the evaluation to a broader range of datasets and programming languages to further validate the generalizability and robustness of our approach.

CRediT authorship contribution statement

Rosmaël Zidane Lekeufack Foulefack: Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Elisabetta Provvedini:** Writing – review & editing, Validation, Software, Methodology. **Alessandro Marchetto:** Writing – review & editing, Validation, Supervision, Software, Project administration, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

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

Acknowledgments

The work was partially supported by the European Union Horizon Europe Research and Innovation Programme under Grant 101120393 (Sec4Ai4Sec) and partially by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU

Data availability

The analysis in this study utilizes well-known public datasets from various sources, such as QEMU+FFmpeg, ProjectKB, Big-Vul, and the CVEFixes dataset. Our source code, along with detailed instructions, is available on GitHub at SCVdet.

References

Bhandari, G., Naseer, A., Moonen, L., 2021. 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.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9), 3280–3296.

Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–33.

Croft, R., Newlands, D., Chen, Z., Babar, M.A., 2021. An empirical study of rule-based and learning-based approaches for static application security testing. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–12.

Do Xuan, C., Quang, D.B., Quang, V.D., 2026. A novel approach for software vulnerability detection based on ensemble learning model. Comput. Electr. Eng. 130, 110848.

Dong, Y., Tang, Y., Cheng, X., Yang, Y., Wang, S., 2023. SedSVD: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. Inf. Softw. Technol. 158, 107168.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. 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.

Feng, H., Fu, X., Sun, H., Wang, H., Zhang, Y., 2020. Efficient vulnerability detection based on abstract syntax tree and deep learning. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, pp. 722–727.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Foulefack, L., Zidane, R., Marchetto, A., 2024a. Enhanced graph neural networks for vulnerability detection in Java via advanced subgraph construction. In: IFIP International Conference on Testing Software and Systems. Springer, pp. 131–148.

Foulefack, L., Zidane, R., Marchetto, A., 2024b. A rapid review on Graph-Based learning vulnerability detection. In: International Conference on the Quality of Information and Communications Technology. Springer, pp. 355–372.

Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.

Grover, A., Leskovec, J., 2016. Node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 855–864.

Hamilton, W., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. Adv. Neural Inf. Process. Syst. 30.

Haque, R., Ali, A., McClean, S., Khan, N., 2025. Explainable vulnerability detection in C/C++ using edge-aware graph attention networks. arXiv preprint arXiv:2507.16540.

Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. LineVD: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 596–607.

Jin, X., Han, J., 2010. K-Means clustering. In: Sammut, C., Webb, G.I. (Eds.), Encyclopedia of Machine Learning. Springer US, Boston, MA, ISBN: 978-0-387-30164-8, pp. 563–564. http://dx.doi.org/10.1007/978-0-387-30164-8_425.

Joern, 2024. Joern version 2.0.331. URL https://joern.io/.

Kaniewski, S., Schmidt, F., Enzweiler, M., Menth, M., Heer, T., 2025. A systematic literature review on detecting software vulnerabilities with large language models. arXiv preprint arXiv:2507.22659.

Li, Q., Liu, W., Wang, Y., Dong, W., 2025. MLAF-VD: A vulnerability detection model based on multi-level abstract features. J. Inf. Secur. Appl. 93, 104189.

Li, Y., Wang, S., Nguyen, T.N., 2021. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303.

Li, Z., Wang, N., Zou, D., Li, Y., Zhang, R., Xu, S., Zhang, C., Jin, H., 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secur. Comput. 19 (4), 2244–2258.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I., 2018. Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118.

Liu, R., Wang, Y., Xu, H., Liu, B., Sun, J., Guo, Z., Ma, W., 2024. Source code vulnerability detection: Combining code language models and code property graphs. arXiv preprint arXiv:2404.14719.

Marchetto, A., Lekeufack Foulefack, R.Z., 2024. Towards the use of domain knowledge to enhance transformer-based vulnerability detection. In: International Conference on the Quality of Information and Communications Technology. Springer, pp. 373–390.

McAfee, 2018. The exactis data breach: What consumers need to know. URL https://www.mcafee.com/blogs/privacy-identity-protection/exactis-data-breach/.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Mirsky, Y., Macon, G., Brown, M., Yagemann, C., Pruett, M., Downing, E., Mertoguno, S., Lee, W., 2023. {$VulChecker$}: Graph-based vulnerability localization in source code. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6557–6574.

Mitre, 2024. 2024 CWE top 25 most dangerous software weaknesses. URL https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html.

Ni, C., Guo, X., Zhu, Y., Xu, X., Yang, X., 2023. Function-Level vulnerability detection through fusing Multi-Modal knowledge. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1911–1918. http://dx.doi.org/10.1109/ASE56229.2023.00084.

OpenAI, 2025. GPT OSS safeguard 120B. URL https://openai.com/index/introducing-gpt-oss-safeguard/.

Peng, X., Wang, S., Qin, Y., Lin, B., Chen, L., Cheng, J., Mao, X., 2025. Keep it simple: Self-Adaptive code graph simplification for accurate vulnerability detection. IEEE Trans. Softw. Eng..

Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C., 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 383–387.

Reimers, N., Gurevych, I., 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084.

securityboulevard, 2025. Threats-breaches. URL https://securityboulevard.com/threats-breaches/.

Shao, M., Ding, Y., Cao, J., Li, Y., 2025. GraphFVD: Property graph-based fine-grained vulnerability detection. Comput. Secur. 104350.

Shiri Harzevili, N., Boaye Belle, A., Wang, J., Wang, S., Jiang, Z.M., Nagappan, N., 2024. A systematic literature review on automated software vulnerability detection using machine learning. ACM Comput. Surv. 57 (3), 1–36.

Song, Z., Wang, J., Yang, K., Wang, J., 2023. HGIVul: Detecting inter-procedural vulnerabilities based on hypergraph convolution. Inf. Softw. Technol. 160, 107219.

Sonnekalb, T., Heinze, T.S., Mäder, P., 2022. Deep security analysis of program code: A systematic literature review. Empir. Softw. Eng. 27 (1), 2.

Tran, H.-C., Tran, A.-D., Le, K.-H., 2025. DetectVul: A statement-level code vulnerability detection for Python. Future Gener. Comput. Syst. 163, 107504.

Wang, S., Huang, C., Yu, D., Chen, X., 2023. VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. Softw.: Pract. Exp. 53 (8), 1631–1658.

Wang, Y., Li, X., Zhang, Y., Li, Y., Zhou, Z., Feng, R., 2025. It only gets worse: Revisiting DL-Based vulnerability detectors from a practical perspective. arXiv preprint arXiv:2507.09529.

Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2275–2286.

Wu, B., Liu, S., Xiao, Y., Li, Z., Sun, J., Lin, S.-W., 2023. Learning program semantics for vulnerability detection via vulnerability-specific inter-procedural slicing. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1371–1383.

Xiao, P., Xiao, Q., Zhang, X., Wu, Y., Yang, F., 2024. Vulnerability detection based on enhanced graph representation learning. IEEE Trans. Inf. Forensics Secur. 19, 5120–5135. http://dx.doi.org/10.1109/TIFS.2024.3392536.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yang, Y., Li, G., 2024. On the code vulnerability detection based on deep learning: A comparative study. IEEE Access.

Yuan, B., Lu, Y., Fang, Y., Wu, Y., Zou, D., Li, Z., Li, Z., Jin, H., 2023. Enhancing deep learning-based vulnerability detection by building behavior graph model. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2262–2274.

Zhang, B., Zhi, X., Wang, M., Ren, R., Dong, J., 2025. Enhancing java web application security: Injection vulnerability detection via interprocedural analysis and deep learning. IEEE Trans. Reliab..

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.