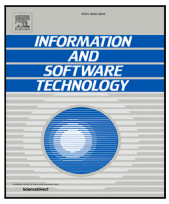
Information and Software Technology 195 (2026) 108104

Contents lists available at ScienceDirect

Information and Software Technology

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



# Domain-aware graph neural networks for source code vulnerability detection

Rosmael Zidane Lekeufack Foulefack *,[1], Alessandro Marchetto [1]

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

## ARTICLE INFO

Dataset link: https://github.com/SCVDetect/VVulDet/

*Keywords:*
Software vulnerability detection
Graph neural network
Domain knowledge
Software security
Source code

## ABSTRACT

**Context:** Deep learning, in particular, graph-based models, has advanced software vulnerability detection by capturing structural code features. However, existing approaches often rely solely on source code and focus mainly on C/C++ at the function level, limiting their ability to detect fine-grained vulnerabilities in diverse languages like Java and Python.
**Objective:** To overcome these limitations, we propose *VVulDet*, an enhanced Graph Neural Network model.
**Methods:** VVulDet enriches code representations with complex graph representations through random walks feature update and incorporates domain knowledge from CVE and CWE descriptions, along with expert-provided reference code fragments. This integration enhances the model's understanding of vulnerabilities beyond pure code structure.
**Results:** We evaluate VVulDet on four datasets covering Java, Python, and C/C++, demonstrating consistent improvements at both statement and function level detection. Notably, VVulDet achieves, on average, the highest overall performance across all datasets, demonstrating F1-score improvements of up to 8.6% and 9.6% at the statement and function levels on ProjectKB, 6.8% and 15.5% on MegaVul, 1.4% and 4.5% on CVEFixes, and 2.4% and 23.7% on BigVul, respectively, compared to the model version that does not incorporate domain knowledge.
**Conclusion:** These results confirm that integrating domain knowledge into graph-based models significantly boosts vulnerability detection performance across multiple programming languages and granularity levels.

## 1. Introduction

Early detection of vulnerabilities in source code is crucial for ensuring software security and protecting systems from potential malicious attacks. Security testing focuses on developing methods and tools to identify software vulnerabilities, thereby reducing the risk of exploitation [1]. Within this domain, software vulnerability detection has emerged as a critical area of research, aiming to create effective tools for uncovering vulnerabilities and weaknesses in source code [2]. Traditionally, this process relied on manual analysis, requiring expert knowledge and significant time investment. As software systems grow increasingly complex, automating vulnerability detection has become indispensable to meet the demands of modern software security.

Conventional static methods for software vulnerability detection analyze source code without requiring its execution [3]. These methods typically rely on predefined rules and source code parsing to identify patterns associated with vulnerabilities [4] or employ techniques such as control and data-flow analysis, including taint analysis [5]. While effective in locating vulnerabilities, these methods often generate a significant number of false positives, requiring manual review. To minimize the need for developer intervention and automate the process, deep learning (DL) and large language models (LLMs) can be utilized [6,7]. Modern methods, named *data-driven techniques*, utilize learning models to analyze and predict vulnerabilities in source code. These models, trained on labeled datasets of vulnerable and non-vulnerable code, can generalize to unseen code. Two main types of methods exist: sequence-based and graph-based learning methods [8].

Sequence-based methods treat source code as a sequence of *tokens*. Fragments of code are split into consecutive tokens and then analyzed by the learned model. For instance, [9,10] developed transformer-based approaches to detect vulnerabilities in C/C++ code. These models require minimal parsing of source code, but their performance may be affected by highly structured, complex, and articulated code, which is difficult to analyze and map into learned patterns, then used to identify vulnerabilities.

* Corresponding author at: University of Trento, Trento, Italy.
*E-mail addresses:* rz.lekeufack@unitn.it (R.Z. Lekeufack Foulefack), alessandro.marchetto@unitn.it (A. Marchetto).
[1] Contributed equally to this research
[2] https://huggingface.co/datasets/DetectVul

https://doi.org/10.1016/j.infsof.2026.108104
Received 1 September 2025; Received in revised form 2 March 2026; Accepted 2 March 2026
Available online 9 March 2026

Graph-based methods leverage graph theory and DL techniques to analyze source code from a structural perspective. These methods convert source code into graph representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), or Program Dependency Graphs (PDGs). Nodes in these graphs represent code constructs like variables (i.e., a named location in a computer's memory that can store data and values), functions (i.e., a reusable fragment of code in a given programming language that performs a specific task), or statements (i.e., a single instruction that the computer can execute in the target programming language, such as an assignment, a control-flow statement, a function call). Edges of graphs capture relationships such as control flow and data dependencies [8]. To detect vulnerability patterns, advanced DL architectures like Graph Neural Networks (GNNs), Gated Graph Sequence Neural Networks (GGNNs), Graph Convolutional Networks (GCNs), and Bidirectional Graph Neural Networks (BGNNs) are employed. Each of these methods processes and aggregates graph information differently, enabling the extraction of both syntactic and structural features from the code. For example, [11] presented LineVD, a technique for vulnerability detection at the statement-level. [12] extended LineVD by adopting Node2vec [13] for generating code elements (e.g., variables and statements) embeddings enriched with contextual information that better capture complex relationships among code elements. [14] developed DetectVul to detect vulnerabilities in Python code at statement-level, and applied it to two datasets: CWEFixes and Vudenc.[2]

This research extends prior work on GNN vulnerability detection [12] by advancing the code representation with the introduction of novel edge types derived from the extraction of Data Dependence Graph (DDG) from the code, thus facilitating the creation of deeper and more expressive graph structures. These enhanced graphs extend beyond conventional Abstract Syntax Trees (AST) and Control Flow Graphs (CFG) for providing a more comprehensive characterization of complex code constructs such as loops and intricate data dependencies. This code representation refinement empowers the GNN to more effectively model critical code relationships, thereby enhancing both vulnerability detection performance and overall software analysis capabilities.

Although data-driven methods have demonstrated their potential in identifying vulnerabilities within source code, they typically rely exclusively on the code as the sole source of information [15]. In contrast, [16] systematically highlighted the limitations of using source code alone as metadata for deep learning models. This reliance often restricts their capacity to grasp a more comprehensive understanding of vulnerabilities, as they focus solely on code-level features during the training. Moreover, many existing vulnerability detection methods are tailored to a given programming language, i.e., C/C++, with limited exploration across other languages [17].

In this work, we extend our preliminary investigation on integrating domain knowledge into GNNs for vulnerability detection [18]. We address the limitations inherent in existing data-driven approaches by enhancing a GNN-based model to identify vulnerabilities across the source code of different programming languages, utilizing not only source code features but also additional contextual information. Our method introduces several feature components that collectively alter the model's detection behavior fundamentally. First, we construct a program graph that unifies source code features from the code property graph with semantic signals, including Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE) descriptions (CWE/CVE text embeddings), as well as exemplar vulnerable code fragments. This integration enriches the model with contextual, semantic, and syntactic information during the learning process of vulnerability identification. This refinement enables the GNN to effectively model critical code relationships, thereby enhancing both vulnerability detection performance and overall software analysis capabilities. The underlying idea is that (i) CVE descriptions and vulnerable code fragments capture established knowledge of known vulnerabilities, and (ii) CWE descriptions and reference vulnerable code fragments represent known code weaknesses that often lead to exploitable vulnerabilities. Hence, the primary goal of the proposed detection method is to assist developers in automatically analyzing their code, enabling early detection of known vulnerabilities and weaknesses, and ultimately helping to prevent the deployment of software containing exploitable security flaws. However, we argue that the use of such semantic information not only allows the model to detect existing vulnerabilities but also leverages its understanding of known weakness descriptions to enhance its capability in identifying potential new vulnerabilities that may share relationships with existing ones.

Compared to our preliminary experimentation, this study extends the previous work in three main directions:

- We enhance the program representation by incorporating DDGs, enabling the model to capture fine-grained data flow and dependency information within the source code.
- The proposed methodology is extended to support multiple programming languages, including Java, Python, and C/C++, and is systematically evaluated on diverse real-world dataset such as ProjectKB, CVEFixes, MegaVul, and BigVul.
- We conduct an extensive experimental study across multiple baselines, including transformer-based and recent large language model-based approaches for source code vulnerability detection.

The experimental results demonstrate significant improvements in identifying vulnerabilities at both the function and statement levels.

## 2. Background

### 2.1. Domain knowledge: Vulnerability and common weaknesses

Software vulnerabilities often arise from inherent weaknesses in code, which can be systematically defined using established knowledge extracted from CVE and CWE [16]. The CVE framework provides a standardized repository of known vulnerabilities observed in specific software versions, assisting developers and security professionals in efficiently identifying and tracking security issues. By referencing CVE entries, organizations can promptly assess potential risks within their software and implement necessary patches, strengthening their defenses against cyberattacks [19]. Table 1 shows an example of a description for the CVE-2020-10221. We can see that several items, e.g., *Description*, *Affected software*, are used to characterize each CVE.

The CWE framework classifies the underlying code weaknesses that may lead to vulnerabilities, offering a structured approach to understand and prioritize security threats. CWEs are organized hierarchically, to represent their inter-dependencies. Using the CWE categories, software engineers gain insight into common coding flaws and their impact, enabling more targeted efforts to improve software quality and security. Table 2 illustrates an example of description of the weakness underlying the CVE presented in Table 1, as outlined by MITRE. Like the CVE, also the CWE is described by several fields, e.g., *Weakness Name* and *Description*. Table 2 reports an example for the CWE-78.

CVE and CWE serve as essential tools in the development of resilient software systems, facilitating both vulnerability management and proactive threat mitigation [16,20]. By using the information from both CVEs and CWEs, we aim at producing a new vulnerability detection method enriched with domain knowledge information that can focus in particular on discovering vulnerabilities and weaknesses that can affect the code, thus avoiding the deployment of software products with known and exploitable vulnerabilities.

### 2.2. Feature embedding

Transforming textual data into a form suitable for machine learning and AI algorithms is a critical step in preparing deep neural networks for training. This transformation involves converting text into numerical formats (e.g., vectors of binary values) that can be interpreted by

**Table 1**
MITRE system: an extract of a CVE description example.

| ID: CVE-2020-10221 | |
|---|---|
| Vulnerability name | rConfig OS Command Injection Vulnerability |
| Affected software | rConfig — Network Configuration Management |
| Date added | 11/03/2021 |
| Required Action | Apply updates per vendor instructions |
| References to Advisories, Solutions, and Tools | Third Party Exploit: https://engindemirbilek.github.io/rconfig-3.93-rce ... |
| Description | lib/ajaxHandlers/ajaxAddTemplate.php in rConfig through 3.94 allows remote attackers to execute arbitrary OS commands via shell metacharacters in the fileName POST parameter. |
| CVSS | Base Score: 8.8 HIGH |
| Weakness Enumeration | CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') |
| Affected Software Configurations | cpe:2.3:a:rconfig:rconfig:3.9.4:*:*:*:*:*:*:* |

**Table 2**
MITRE system: MITRE system: an extract of a CWE description example.

| ID: CWE-78 | |
|---|---|
| Weakness Name | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') |
| Description | The product constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component. |
| Scope | Confidentiality, Integrity, Availability, Non-Repudiation |
| Technical Impact | Execute Unauthorized Code or Commands; DoS: Crash, Exit, or Restart; Read Application Data; Modify Application Data; Hide Activities |
| Relationships: ChildOf | CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection') CWE-74 Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection') |
| Likelihood Of Exploit | High |
| Reference code | This code intends to take the name of a user and list the contents of that user's home directory. It is subject to the first variant of OS command injection.<br>`&userName = &_POST["user"];`<br>`&command = 'ls␣-l␣/home/' . &userName;`<br>`system(&command);`<br>The `$userName` variable is not checked, an attacker could set the `$userName` variable to an arbitrary OS command such as: `';rm -rf /'` which results in $command: `'ls -l /home/;rm -rf /'`. In Unix, a semicolon is a command separator; the OS first executes ls and then rm. |

algorithms. The process typically begins with tokenization, where the text (code in our case) is divided into smaller components known as *tokens*. Subsequently, various embedding techniques can be used to represent the text as vectors at different granularities, such as word, sentence, or document levels.

In the context of source code, several methods for software vulnerability detection have leveraged embedding approaches like GloVe,[3] Doc2vec,[4] Word2vec,[5] and CodeBERT[6] to generate pre-trained token vectors. For example, [11,21] examined multiple embedding strategies and demonstrated that the selected embedding method can significantly influence a model's performance.

While these approaches generate embeddings at varying levels, the embeddings' nature differs considerably. Static embedding methods, such as Word2vec, GloVe, and FastText,[7] assign a fixed vector to each token regardless of its context. In contrast, contextual embedding models like CodeBERT and Doc2vec produce dynamic representations, where token embeddings consider the surrounding text [11]. In this research, we utilized CodeBERT model for feature extraction. However, all existing embedding techniques, which map input data into a continuous vector space using linear transformations, often fail to capture the nonlinear relationships inherent in code. For example, techniques like Glove represent code tokens based solely on statistical occurrences, lacking the contextual understanding required for effective vulnerability detection. Hence, more recent ones, such as GraphSAGE and Node2vec, are designed for graph-based architectures and generate contextual embeddings that account for complex relationships between neighboring nodes of the graph representing both structure and content of the input text (source code in our case). In this work, we use Node2vec to better capture the structural and topological characteristics of source code when it is represented by a graph-base structure, with the aim of enhancing model comprehension and performance.

### 2.3. GNN for vulnerability detection

Graph-based models have demonstrated superior performance with respect sequence-based models in code-related tasks such as code clone detection [22] and code classification [9,23]. These models effectively capture intricate semantic and syntactic relationships within source code elements (e.g., variables and statements). The process involves parsing the code to construct a Code Property Graph (CPG), a versatile data structure tailored for analyzing large codebases.[8] In a CPG, nodes typically represent entities such as functions and statements, while edges encode program dependencies like control flow, data flow, or call

[3] http://nlp.stanford.edu/projects/glove/
[4] https://github.com/jhlau/doc2vec
[5] https://code.google.com/archive/p/word2vec
[6] https://github.com/microsoft/CodeBERT
[7] https://github.com/benathi/multisense-prob-fasttext
[8] https://docs.joern.io/code-property-graph

relationships. Mathematically, a graph is defined as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of nodes, and $\mathcal{E}$ denotes the set of edges.

To enable GNNs to uncover meaningful patterns, textual features of the code (e.g., function structure, control/data flow relationships) are embedded as attributes on nodes and edges, as feature vectors. This integration allows GNNs to capture hidden relationships within the source data. The learning process typically involves three fundamental phases: *message passing*, *aggregation*, and *update*.

- *Message Passing:* In this phase, each node exchanges information with its neighbors. For a node $v \in \mathcal{V}$, the message passed from a neighboring node $u$ is expressed as:

$$m_{u \to v}^{(t)} = \phi(h_u^{(t-1)}, h_v^{(t-1)}, e_{uv}), \tag{1}$$

  where $h_u^{(t-1)}$ and $h_v^{(t-1)}$ are the feature vectors of nodes $u$ and $v$ at the previous time step $t-1$, $e_{uv}$ represents the edge features between nodes $u$ and $v$, and $\phi$ is a predefined or learnable function.
- *Aggregation:* Each node aggregates the messages received from its neighbors into a single representation. For a node $v$, this operation is defined as:

$$a_v^{(t)} = \psi(\{m_{u \to v}^{(t)} \mid u \in \mathcal{N}(v)\}), \tag{2}$$

  where $\mathcal{N}(v)$ represents the set of neighbors of $v$, and $\psi$ is an aggregation function, such as summation, mean, or maximum pooling.
- *Update:* Finally, the node updates its feature vector using the aggregated information. It is expressed as:

$$h_v^{(t)} = \gamma(h_v^{(t-1)}, a_v^{(t)}), \tag{3}$$

  where $\gamma$ is a learnable update function, often implemented as a feedforward neural network.

These steps are iteratively applied over multiple layers and time steps, enabling GNNs to propagate information across the graph and learn increasingly global representations of its structure.

Graph-based learning techniques allow GNNs to utilize the hierarchical and non-linear characteristics of source code, surpassing the capabilities of sequence-based models in detecting low-level details. Furthermore, advanced GNN architectures, such as Graph Attention Networks (GATs) and GraphSAGE, refine the learning process by incorporating mechanisms like attention-based message weighting (to assign a numerical value to different parts of an input, with the aim of giving different relevance to such parts in a given context) and sampling-based neighborhood aggregation (instead of aggregating information from all of a node's neighbors, only a subset of them is used with the aim of making the approach more scalable).

## 3. Methodology

In this work, we aim at advancing vulnerability detection in source code by building on an existing tool originally designed for C/C++. We opted for LineVD as initial model due to its promising performance in detecting vulnerabilities at the statement level. Unlike many graph-based approaches that primarily make function-level predictions, requiring additional processing to pinpoint the specific vulnerability-related statements inside each function, the model's architecture is tailored for fine-grained detection. However, the original model focuses only on C/C++ code, and it faces limitations in capturing long-range dependencies within the code, thus limiting its applicability and performance. To address these issues, we implemented an evolved version that: (i) works for several programming languages (C/C++, Java, and Python); (ii) introduces a deep graph-based representation of the source code that includes AST, CFG, PDG, DFG, and DDG to better account for complex and long-range relationships among code statements; (iii) uses Node2Vec to generate better contextualized embeddings for code elements and statements, i.e., nodes of the graph-based representation; and (iv) adds additional sequential layers to process extra information, with the aim of incorporating domain knowledge for vulnerability detection.

### 3.1. Definition

We formulate statement-level vulnerability detection as a binary node classification task. Let ($V_i$, $Y_i$) represents a data sample, where $V_i \in V$ corresponds to a program statement, and $Y_i \in \{0, 1\}$ denotes its label (i.e., 0 indicating non-vulnerable and 1 indicating vulnerable). For each $i \in \{1, 2, \ldots, n\}$, where $n$ is the total number of nodes in the dataset, the objective is to train a model that maps each statement $V_i$ to its corresponding label $Y_i$, as expressed by:

$$f : V \to Y, \tag{4}$$

where $V$ represents the set of all nodes, and $Y = \{0, 1\}$ is the set of possible labels. The function $f$ is optimized by minimizing the cross-entropy loss, which can be formulated as:

$$\min \sum_{i=1}^{n} C(f(G_i, Y_i | V_i)), \tag{5}$$

where $G$ denotes the graph structure used for modeling the relationships between nodes. The cross-entropy loss quantifies the difference between the true label distribution and the predicted probability distribution, with the objective of reducing this discrepancy during training.

The loss function, when considering a data pair $(x, y)$, where $x$ represents the input features and $y$ the target labels, can be written as:

$$l(x, y) = L = \{l_1, l_2, \ldots, l_N\}^T, \tag{6}$$

where

$$l_n = \sum_{k=1}^{K} \omega_k \, y_{n,k} \log \frac{\exp(x_{n,k})}{\sum_{i=1}^{K} \exp(x_{n,i})}. \tag{7}$$

In this formulation, $\omega_k$ represents the class weight, $K$ is the number of classes, and $N$ refers to the size of a mini-batch, which is a subset of training samples processed together during training. The cross-entropy loss in (7) computes the weighted log-likelihood of the true class $y_{n,k}$ based on the predicted probabilities, which are obtained via a softmax transformation of the model's raw output scores.

### 3.2. Model architecture

The proposed vulnerability detection method is executed in three main phases: (i) **Data collection and cleaning**, a preliminary step required to collect and clean the needed data and information (e.g., labeled source code, and domain knowledge information about CVEs and CWEs — see Section 2.1); (ii) **Data + Domain Knowledge**, the source code fragments in the training dataset are analyzed, a graph-based representation is extracted for each fragment, and the domain knowledge information is incorporated; and (iii) **Model Training and Evaluation**, the vulnerability detection model is trained and then used to detect the presence of vulnerable code in previously unseen code fragments (e.g., the ones contained in the test dataset).

Fig. 1 provides an overview of the model architecture that supports the proposed method. The architecture comprises three key components: (1) *feature extraction*, (2) *graph construction*, and (3) *classification*.

1. **Feature Extraction:** By starting from the source code and the collected information about CVEs and CWEs (both descriptions and reference code fragments), we utilize a transformer-based approach for tokenizing both the text of the CVE and CWE descriptions and the reference code. By applying CodeBERT's Byte-Pair Encoding tokenizer [24], we generate contextualized embeddings for each collected element, then pass them as input to the next component.

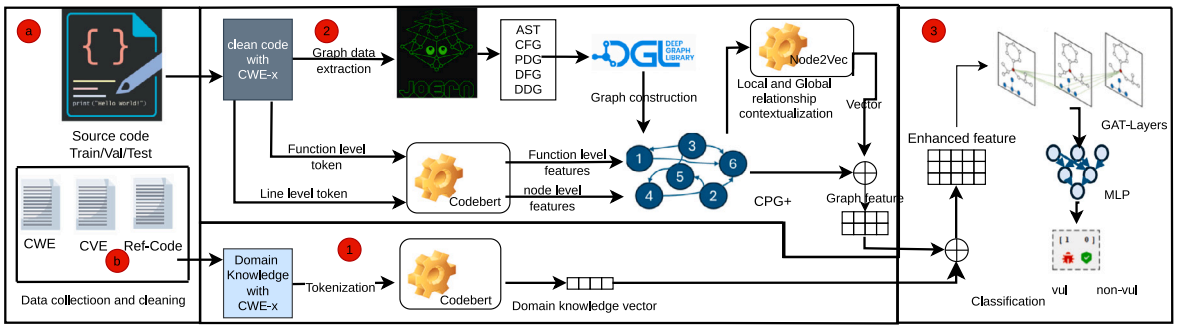



**Fig. 1.** Overview of the Proposed Framework: (a) source code, (b) domain knowledge, 1: Feature Extraction, 2: Graph Construction, and 3: Classification.

2. **Graph Construction and feature:** From the source code, we extract several graph-based representations, e.g., AST, CFG, PDG. In this work, we additionally introduce the Data Dependence Graph (DDG) for further improving the representation of complex relationships within source code elements (e.g., variables and code statements). All these graph-based representations, extracted with the tool Joern, are then joined into an enriched Code Property Graph (named *CPG+*). The constructed CPG+ embeddings extracted with CodeBERT are further enriched by incorporating contextualized embeddings from Node2Vec, capturing both local and global code dependencies. Node2Vec is a graph integration algorithm that learns to represent nodes in low dimensions by capturing local and global structural information through random walks. Node2Vec uses biased random walks to explore the graph, balancing breadth-first and depth-first search strategies to preserve relationships between nodes. This approach improves the model's ability to handle noisy data and long-term dependencies by encoding subgraph structures more effectively. The output of this stage consists of graph-based feature embeddings, which are concatenated with the embeddings from the previous stage (output of Feature Extraction), following a mean pooling operation. Process by an additional layer before aggregation, this combination aims to produce code embeddings enriched with contextual and domain knowledge. To ensure compatibility and support efficient computation in subsequent attention layers, a linear transformation layer is introduced. This layer learns to project the concatenated embeddings into a unified space, addressing the issue of varying input dimensions that can arise during message parsing and feature aggregation.
   Formally, let $B$ be an $n$-dimensional vector space over $\mathbb{R}$. Each text element $te \in T$ (e.g., statement or function description) is mapped to a vector $b \in B$ using an embedding function $g$:

$$b = g(te). \tag{8}$$

   The resulting vectors $b$ are integrated as additional graph features, leveraging the contextualized representations from CodeBERT to improve vulnerability detection.
3. **Classification:** A Graph Attention Network (GAT) is used to learn the topological dependencies within the constructed graph. The GAT layer processes the embedded statements from CodeBERT, along with graph edges, using self-loops through an iterative aggregation process of neighborhood features. The attention mechanism updates node embeddings as follows:

$$z_i^{(t)} = W^{(t)} h_i^{(t)}, \tag{9}$$

$$e_{i,j}^{(t)} = \text{LeakyReLU}\left(\vec{a}^{(t)^T}\left(z_i^{(t)} \parallel z_j^{(t)}\right)\right), \tag{10}$$

$$\alpha_{i,j}^{(t)} = \frac{\exp(e_{i,j}^{(t)})}{\sum_{p\in\mathcal{N}(i)} \exp(e_{i,p}^{(t)})}, \tag{11}$$

$$h_i^{(t+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{i,j}^{(t)} z_j^{(t)}\right), \tag{12}$$

   where $h_i^{(t)}$ represents the node embedding vectors at layer $t$, $W^{(t)}$ and $\vec{a}$ are learnable parameters, $\mathcal{N}(i)$ denotes the neighbors of node $i$, and $\sigma$ is the activation function.
   Our classification approach integrates embedded features at both function and statement levels for decision-making. A multilayer perceptron processes these features, ensuring consistency between predictions at both levels through element-wise multiplication. This design enables the detection of vulnerabilities at various levels of code structure, leading to more accurate predictions.

### 3.3. Model enhancements

Our model, named **VVulDet**, is improved with respect to the initially investigated model in two main directions.

(1) **Enriched CPG (CPG+)**. We believe that the original model did not effectively capture the complexity of the source code, so we updated the extraction process and introduced the use of Data Dependence Graphs (DDGs), which provide deeper structural and semantic insights into code by representing the flow of data within a program. A DDG is a directed graph that captures dependencies between program statements or expressions (nodes of the graph) based on how data is defined, used, and propagated (edges of the graph). It serves as a fundamental abstraction that makes explicit the relationships between variables and operations within the code. Often, DDGs are used in security to study the flow of tainted variables (e.g., variables for which the values are provided by the user) in the code, aiming to identify if such variables are used in potentially vulnerable statements (e.g., SQL injection). The use of DDGs introduces new edge types in our overall graph-based representation of the source code (input for the learning phase), leading to the construction of a deeper graph representation. For instance, given s1 (e.g., "$var1 = 10;$") and s2 (e.g., "$var2 = var1+10;$"), two consecutive code statements in a function inside of the learning dataset (i.e., two consecutive nodes of the overall graph-based representation of the function code), we can observe that there exists a relation s1 → s2 (modeled in the function CFG, capturing the execution flow, i.e., $s1$ is executed before $s2$) and a second relation s1 ← s2 (modeled in the function DDG, capturing the dependency of $s2$ on $s1$). Hence, by composing the overall graph of the function with information from different graph types, such as CFG and DDG, the overall graph inherits the relationships between these two code entities. These edge types are not treated differently during the model's learning process; instead, they affect the node feature refinement through the random walk process using Node2Vec, which refines the node features by considering the topological neighborhood relationships. This graph enhancement enables the model to capture relationships in the code more effectively, ultimately improving its ability to represent complex code structures,

such as loops (graph heterogeneity) and intricate data dependencies, thus increasing the model's capability in detecting vulnerabilities and performing software analysis.

(2) **Domain knowledge incorporation**. The incorporation of expert knowledge begins with gathering all relevant data sources, including domain-specific insights. To achieve this, we collected source code from well-established datasets often used to train learning algorithms for vulnerability detections, such as Big-Vul, MegaVul, CVEFixes, and ProjectKB (Fig. 1a). We then developed a Python script to iterate through all the web pages of the MITRE system, collecting samples of code fragments provided by MITRE experts that describe CWEs. Additionally, we gathered CWE and CVE descriptions from the MITRE system (Fig. 1b). Tables 1 and 2 show examples of CVE and CWE descriptions, such as the ones that we consider, with an example of a small reference code fragment for the CWE. We analyzed both natural language descriptions and reference code fragments for samples across different programming languages in the datasets used. This domain-specific information is then integrated with the training data by aligning unique identifiers shared between the MITRE database and our dataset (e.g., CVE and CWE identifiers). During training, traditional features, such as source code embeddings and external domain knowledge (e.g., vulnerability descriptions and taxonomy from CVE and CWE), are incorporated into the graph feature using a pooling operation to produce a unified feature representation.

For each code sample associated with a specific CWE (e.g., CWE-78 in the dataset), we directly use its official description from MITRE (for example, the CWE-78 description of Table 2). While a few samples without CWE/CVE identifiers are discarded during preprocessing, for CWE/CVE identifiers with missing or limited reference code matches (as summarized in Table 10), we randomly select a reference code from the MITRE repository in the same programming language, since sufficient CWE/CVE textual descriptions are collected from MITRE. This process ensures that each training sample maintains a consistent set of feature components during model training. Let $h$ (12) denote the source code feature, and $b$ the embedding of the textual domain description obtained via the pretrained text encoder, CodeBERT, of the CWE-78 of Table 2 as described by (8). The joint representation is computed by (13) to update the leaning step.

$$h' = l(W_h h + W_b b), \tag{13}$$

where $W_h$ and $W_b$ are learnable weights, while $l(\cdot)$ is a linear concatenation layers. New linear layers (i.e., $l(\cdot)$) are then applied to project the combined features (concatenated) into a consistent embedding space, ensuring compatibility with downstream components of the model. Additionally, a reference source code snippet, drawn from MITRE and corresponding to the same vulnerability class, was introduced as an auxiliary input. A cosine similarity score is computed between the primary source code representation and the reference vulnerable code. This similarity coefficient is then used to weight the global feature vector, thereby enhancing the model's sensitivity to vulnerability-relevant patterns during training. All these updates to the model structure result in a more robust and complex training tool with more trainable parameters compared to the original version. This design choice is motivated by the fact that linear fusion is often preferable when auxiliary modalities correspond to global descriptors, as it enables effective integration while maintaining model simplicity and stability.

## 4. Experimental design

Hereafter, we use: (i) VVulDet$_{noDDG_noDK}$ for our tool and model used in the setup of the preliminary version extended in this work; (ii) VVulDet$_{noDK}$ for the setup with the DDG, but without the domain knowledge information; and (iii) VVulDet for the complete version.

### 4.1. Research question

To assess the effectiveness of the proposed method, the following research questions (RQs) are investigated:

**RQ1:** *How does VVulDet$_{noDK}$ (i.e., without domain knowledge) perform compared to existing sequence and graph-based baselines in vulnerability detection at both the statement and function level for C/C++ code?* This question compares the performance of VVulDet$_{noDK}$ by applying it to a representative C/C++ dataset for direct comparison with existing sequence and graph-based baseline methods. In this question, we focus on C/C++ since most of the state-of-the-art tools are designed specifically for such a programming language.

**RQ2:** *How does VVulDet$_{noDK}$ (i.e., without domain knowledge) perform in detecting vulnerabilities at statement and function level for different programming languages?* This question examines the model's ability to generalize by evaluating its effectiveness on datasets from different programming languages, including Java, C/C++, and Python.

**RQ3:** *What is the impact of integrating domain-specific information on the VVulDet's performance?* This question investigates the benefits of incorporating domain knowledge, such as CVE and CWE descriptions, and vulnerable reference code provided by MITRE, and compares the results with the ones of models that rely solely on source code.

**RQ4:** *How does VVulDet perform compared to state-of-the-art sequence-based models that natively adopt domain knowledge information and LLM for vulnerability detection?* This question investigates the advantages of graph-based models enriched with domain-specific information in comparison to sequential models that also integrate domain knowledge and LLMs. We utilize various variants of sequence-based models that employ language models, such as BERT (specifically JavaBERT), Llama 3.2, Claude-Sonnet-3, Llama-3.1-8B-Instruct, Mistral-7B-Instruct, and LineVul as baselines.

### 4.2. Datasets

We utilized several datasets from the existing literature. Specifically, we used two Java datasets, ProjectKB [25] and MegaVul [26], a C/C++ dataset, Big-Vul [27], and a Python version of CVEFixes [28]. We selected these datasets because they have been used in similar studies, and they contain all the relevant information we need for our tools, i.e., fragments of functions' code and their labels (vulnerable or non-vulnerable), underlying CWEs and referenced CVEs, if any, the code or a reference to a repository with the code used to fix the vulnerability. Table 3 details the distribution of the functions contained in each dataset, their lines of code, and the number of different code weaknesses (CWEs), thus providing a comprehensive overview of the utilized data.

These datasets collect and label source code from a variety of real-world projects. While originally labeled at the function-level, we build upon prior research to derive line-level labels. Specifically: (1) lines removed in a vulnerability-fixing commit are identified as indicative of vulnerabilities, and (2) lines that are control or data dependent on the newly added lines in such commits are also considered as related to the vulnerability. This stems from the understanding that added lines in a vulnerability-fixing commit are designed to address the vulnerability. Consequently, lines that were not directly modified but are related to these additions are also linked to the vulnerability.

To limit the negative impact of working with imbalanced datasets, as observed from Table 3, we implemented a random down-sampling of the non-vulnerable training functions, retaining 20% more samples than those from the vulnerable category. We posit that this approach is more viable, as it ensures that the number of training samples remains substantial for our learning-based model. By adopting this down-sampling strategy for the majority class, we aim to preserve the quality of the dataset and its real-world applicability, rather than incorporating augmented or duplicated source code samples that may lead to overfitting. This process avoids the costly process of generating

**Table 3**
Basic statistical information about the dataset provided at both the function and statement levels. Ratio refers to the proportion of vulnerable samples compared to non-vulnerable samples.

| Dataset | Language | Function | | | Statement | | | Number of CWE |
|---|---|---|---|---|---|---|---|---|
| | | Total | Vulnerable | Ratio(%) | Total | Vulnerable | Ratio(%) | |
| ProjectKB | Java | 20 155 | 2420 | 13.64 | 91 052 | 1826 | 2.04 | 64 |
| Megavul | Java | 41 949 | 2433 | 6.15 | 377 083 | 3759 | 1.01 | 93 |
| CVEFixes | Python | 29 597 | 3305 | 12.57 | 646 109 | 4128 | 0.643 | 121 |
| Big-Vul | C/C++ | 188 636 | 10 900 | 6.33 | 1 016 135 | 24 103 | 2.43 | 91 |

synthetic code with tools such as LLMs, which would require expert validation to ensure that the generated samples are truly vulnerable and might compromise data integrity. However, in the parameter search section, we conducted a series of experiments using the baseline model (i.e., **VVulDet**$_{noDK}$) without domain adaptation, testing different levels of class imbalance in the training data. Here, 100% imbalance refers to using the full training dataset without any balancing procedures. The analysis reveals that the model's performance is more stable when the minority class is 20% less than the non-vulnerable one at the function level, resulting in well-balanced metrics at the statement level. Table 4 shows the performance metrics from these tests, where the PRAUC (a metric that expresses the tradeoff between precision and recall) is higher, highlighting the model's ability to capture the correct vulnerable line.

Although we still encountered a significantly unbalanced dataset at the statement level, we believe that this methodology enabled us to maintain an adequate volume of samples for training our algorithms effectively.

Additionally, a preprocessing step is applied to ensure the quality of the dataset and remove noisy information. We remove comments, discard improperly truncated and duplicated functions, normalize the code (e.g., whether more code statements were identified in the same line of code, we split it into more lines, one per statement), and check a subset of randomly selected code function labels for removing wrong labels. Each considered dataset is then shuffled and split randomly into training, validation, and test sets, using an 80:10:10 ratio.

### 4.3. Evaluation metrics

We evaluate the classification performance using several performance metrics: Precision, Recall, F1-score, and PRAUC (Precision-Recall Area Under the Curve),

- **F1-score:** Balances precision and recall, defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{14}$$

- **Precision:** Measures the proportion of true positives among all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{15}$$

- **Recall:** Calculates the true positive rate:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{16}$$

- **PRAUC** summarizes the trade-off between precision and recall at different thresholds, providing an overall measure of model performance in imbalanced datasets.

$$PRAUC = \int_0^1 \text{Precision}(t) \cdot \text{Recall}'(t)\, dt \tag{17}$$

True Positive (TP) are correctly identified vulnerable code fragments (e.g., entire functions, or specific vulnerable statements inside them), True Negatives (TN) are code fragments correctly identified as non-vulnerable, False Positives (FP) are misclassified non-vulnerable code fragments, and False Negatives (FN) are vulnerable code fragments not identified.

**Table 4**
Imbalanced training set effect on the model performance at statement-level on the ProjectKB dataset.

| Imbalance(%) | Precision | F1-score | Recall | PR-AUC |
|---|---|---|---|---|
| 100 | 0.485 | 0.492 | 0.498 | 0.344 |
| 60 | 0.677 | 0.505 | 0.506 | 0.425 |
| 40 | 0.819 | 0.513 | 0.510 | 0.485 |
| 20 | 0.697 | 0.562 | 0.540 | 0.531 |

**Table 5**
Hyper-parameters of the VVulDet configuration.

| Param. | Ray value | Systematic trials | | | |
|---|---|---|---|---|---|
| Lr | $1e-3$ | $1e-3$ | $2e-6$ | $1e-4$ | $1e-5$ |
| BS | 1024 | 256 | 512 | 1024 | 512 |
| HD | 0.19 | 0.5 | 0.5 | 0.5 | 0.2 |
| GAT D | 0.2 | 0.3 | 0.3 | 0.3 | 0.2 |
| MLP D | 0.19 | 0.2 | 0.2 | 0.2 | 0.2 |
| Epochs | 200 | 30 | 200 | 130 | 250 |

### 4.4. Implementation and parameter selection

A tool that supports the proposed approach has been made available online on GitHub[9]

We conducted our experiments by running the tool on a local server equipped with AMD RyzenTM 9 7950X 16 Core (CPU), 128 GB DDR5 (RAM), NVIDIA GeForce RTX 4090 24 GB GDDR6X (GPU). Table 5 summarizes the configuration of the tool used in the experiment by reporting the relevant parameters. The "Ray values" refer to the set of parameter values obtained through the Ray Tune library[10] for automatic hyperparameter tuning, utilizing grid search on the ProjectKB dataset starting from random lists. These values, along with systematically selected parameters, are applied across each dataset. In the table, "systematic trials" denote additional parameter values adjusted based on the model's performance during the fine-tuning process. We did not fine-tune these parameters differently to maintain the integrity of the evaluation for each dataset and to keep track of the usefulness of domain features. The key modified parameters include learning rate (LR), batch size (BS), hidden layer dropout (HD), GAT layer dropout, MLP layer dropout, and the number of epochs.

### 4.5. Process

We conducted our experiment by applying the following steps to each dataset:

1. Pre-processing the dataset to clean, down-sampling, and randomly split it into training, validation, and testing (Section 4.2);
2. Collecting domain knowledge information for the code contained in the dataset (Section 3.3);
3. Training the models of interest with and without the incorporation of the domain knowledge to detect the vulnerable code, at both function and statement level;

[9] https://github.com/SCVDetect/VVulDet
[10] https://docs.ray.io

**Table 6**
Baseline characterization.

| Baseline | Year | Architecture Learning Architecture | Classifier | Granularity | Feature representation Code Representation | Vectorization Embeddings | Information Source |
|---|---|---|---|---|---|---|---|
| Devign | 2019 | GGRL | 1D Conv, Sigmoid | Function | AST,CFG,DFG,NCS | Word2vec | Code |
| IVDetect | 2021 | FA-GCN | Softmax | Statement | PDG,AST,Flat code | GloVe,GRU | Code |
| LineVD | 2022 | GAT | MLP,ReLU | Statement | PDG,CDG,Flat code | CodeBERT | Code |
| LineVul | 2022 | Transformer | Linear | Function | Flat Code | CodeBERT | Code |
| MVulD | 2023 | GAT,GCN Transformer | Softmax | Function | AST,CFG,PDG, Image,Flat code | UniXcoder | Code |
| SVDet | 2024 | Transformer | Linear | Statement | Flat code | JavaBert | Code |
| eSVDet | 2024 | Transformer | Linear | Statement | Flat code | JavaBert | Domain |
| StagedVulBERT | 2024 | Transformer | DNet | Statement | Flat Code | CodeBERT-HLS | Code |
| RoS-Dex | 2026 | Transformer | MLP | Function | Flat Code | CodeT5+ | Code |

4. Applying the trained models to the test dataset to predict the presence of vulnerable code;
5. Measuring the performance metrics on the obtained predictions. To ensure a fair evaluation, we averaged the metrics of five runs for each new model configuration;
6. Conducting further analysis on collected results.

**Baseline:** We considered both graph-based and transformer baseline models from the literature, including models such as LineVD [9] and IVDetect [23], as well as LineVul [9], StagedVulBERT [29] and RoS-Dex [30]. Additionally, we examined models at the function level, including IVDetect, LineVD, MVulD [31], and Devign [32].

Table 6 summarizes the main characteristic of the baselines. We can see that these models extract different types of graph-based representations from the code (e.g., Abstract Syntax Tree — AST, Control Flow Graph — CFG), adopt different learning architectures (e.g., Graph Attention Networks — GAT, Graph Convolutional Network — GCN, Feature-Attention Graph Convolutional Network — FA-GCN, Gated Graph Recurrent Layers — GGRL, and Transformer), and different feature embedding methods (e.g., UniXcoder, GloVe, Code/JavaBert, Word2vec).

Additionally, to compare different architectures (i.e., graph-based and sequence-based models) while incorporating domain-specific information, we compared our method with eSVDet [33], a transformer model based on JavaBert (a BERT model specialized for Java source code) that applies various mechanisms to integrate the domain knowledge information into a transformer model named SVDet [34] that works for Java code at both statement and function level. Moreover, we also considered LLM-based vulnerability detection approaches, such as Claude-Sonnet-3, Llama 3.2, Llama-3.1-8B-Instruct, and Mistral-7B-Instruct, state-of-the-art large language models capable of understanding and reasoning about source code. Claude[11] and Llama[12] leverage their pretrained knowledge of software patterns and natural language semantics to detect potential vulnerabilities without requiring explicit feature engineering. To facilitate this, we provided each code snippet from the test set along with a prompt asking models to identify whether the code contains a vulnerable line (keeping models' Temperature equal to 0.5). On the other hand, Llama-3.1-8B-Instruct and Mistral-7B-Instruct are open-source models accessible from the Hugging Face[13] platform, fine-tuned following the methods in [35], and queried similarly to the other two.

[11] www.claude.ai
[12] https://ai.meta.com/llama
[13] https://huggingface.co/models

## 5. Results

### 5.1. RQ1 — $VVulDet_{noDK}$ compared to existing graph and transformer-based baselines for C/C++

We compare two variants of the proposed model, namely $VVulDet_{noDDG\_noDK}$ and $VVulDet_{noDK}$. The former variant [12] enhances LineVD by employing Node2Vec [13] to generate contextual embeddings that better capture complex local relationships among code elements (e.g., variable dependencies). This variant serves as the basic version of our approach, as it does not integrate DDG edges into the code graph representation. In contrast, the latter variant incorporates both Node2Vec features and DDG edges into the graph-based representation of the source code, enabling deeper connectivity between nodes in the graph. These variants, trained on the C/C++ dataset BigVul, are compared with other graph-based baselines, namely IVDetect, MVulD, Devign, and LineVD. The focus on C/C++ reflects its widespread use in vulnerability detection research and the availability of established tools for this language. Among the baselines, MVulD and Devign are designed for function-level detection only, while LineVD and IVDetect provide predictions at both statement and function levels. Table 7 reports the results.

At the statement level, while LineVD outperforms the other baselines in terms of PRAUC (+0.121), $VVulDet_{noDK}$ achieves higher precision, F1-score, and recall (+0.85, +0.190, and +0.147, respectively). IVDetect and LineVD trail significantly in precision and F1-score (up to a maximum of 0.271 for precision and of 0.360 for F1-score), with IVDetect having notably lower recall (only 0.140 with respect to the 0.542 of our method). Additionally, in a manual evaluation of randomly selected graphs, we noticed structural differences, such as variations in the number of nodes and edges, when including or excluding DDG edges, thereby highlighting the importance of considering DDG information in graph construction. The improved results stem from $VVulDet_{noDK}$'s use of contextual node and edge embeddings with Node2vec and the integration of DDG edges capturing long-range code dependencies.

At the function level, $VVulDet_{noDDG\_noDK}$ outperforms $VVulDet_{noDK}$ for the precision (+0.009) but $VVulDet_{noDK}$ is still competitive, and it outperforms in precision the other baselines (on average, +0.33) and in F1-score, recall, and PRAUC all baselines (on average, +0.262, +0.249, +0.078, and +0.171 respectively). Recall is high and comparable for $VVulDet_{noDK}$ (0.722) and IVDetect (0.720), both slightly higher than LineVD (0.713).

Compared to recent transformer-based models like LineVul, StagedVulBERT, and RoS-Dex (see Table 8), our model ($VVulDet_{noDK}$) achieves higher recall and F1-score than RoS-Dex, although it falls short in precision, which reaches 0.710. In contrast, StagedVulBERT

**Table 7**
Metrics for statement and function level on Big-Vul (best in **bold**) compared to Graph-based models. "NA" means the model is not designed for that granularity.

| Model | Language | Statement-level | | | | Function-level | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Prec | F1-score | Rec | PRAUC | Prec | F1-score | Rec | PRAUC |
| IVDetect | C/C++ | 0.238 | 0.176 | 0.140 | 0.520 | 0.236 | 0.355 | 0.720 | 0.479 |
| LineVD | C/C++ | 0.271 | 0.360 | 0.533 | **0.642** | 0.542 | 0.615 | 0.713 | 0.617 |
| MVulD | C/C++ | NA | NA | NA | NA | 0.183 | 0.272 | 0.531 | 0.357 |
| Devign | C/C++ | NA | NA | NA | NA | 0.090 | 0.157 | 0.617 | 0.354 |
| VVulDet$_{noDDG_noDK}$ | C/C++ | 0.505 | 0.492 | 0.510 | 0.510 | **0.602** | 0.610 | 0.638 | 0.621 |
| VVulDet$_{noDK}$ | C/C++ | **0.523** | **0.533** | **0.542** | 0.533 | 0.593 | **0.651** | **0.722** | **0.657** |

**Table 8**
Performance of our model compared to sequence-based models on the Bigvul dataset, analyzed at both the statement and function levels.

| Granularity | Model | Metrics | | | |
|---|---|---|---|---|---|
| | | Top-2% | Top-4% | Top-6% | Top-10% |
| Statement | StagedVulBERT | **63.1** | **64.0** | **64.2** | **65.1** |
| | LineVul | 31.1 | 35.0 | 38.8 | 44.8 |
| | VVulDet$_{noDK}$ | 27.58 | 33.35 | 40.1 | 48.76 |
| | VVulDet$_{CVECWE}$ | 29.5 | 35.6 | 41.2 | 50.1 |
| | | Pre | F1-score | Rec. | PRAUC |
| Function | StagedVulBERT | **0.943** | **0.923** | **0.903** | – |
| | LineVul | 0.942 | 0.856 | 0.785 | – |
| | RoS-Dex | 0.710 | 0.613 | 0.539 | – |
| | VVulDet$_{noDK}$ | 0.593 | 0.651 | 0.722 | 0.657 |
| | VVulDet$_{CVECWE}$ | 0.893 | 0.888 | 0.883 | **0.810** |

outperforms our model at both metric levels, highlighting the advantages of recent transformer designs over a graph-based approach. In top-k% evaluations at the statement level, VVulDet$_{noDK}$ demonstrates comparable performance to LineVul, slightly surpassing it at k=6 and 10. Furthermore, the enhanced version investigated in this paper that incorporates CVE and CWE domain descriptions (VVulDet$_{CVECWE}$) shows improved performance over VVulDet$_{noDK}$ and LineVul by starting from top-4%, achieving up to 50.1% accuracy at $k = 10$. Additional domain improvements for our model are detailed in Section 5.4

These observations demonstrate that VVulDet$_{noDK}$ has a more balanced trade-off between precision and recall, leading to superior overall effectiveness compared with graph-based baselines. Its design, which aggregates statement-level detections while enriching contextual representations, contributes to its robustness. Although some baselines, particularly the sequence-based transformer StagedVulBERT, achieve higher performance, our model remains competitive with ensemble-based transformer approaches such as RoS-Dex at the function level and LineVul at statement level.

**RQ1:** By leveraging Code Property Graphs that include DDG edges, and by contextualizing graph embeddings with Node2Vec, VVulDet$_{noDK}$ outperforms graph-based state-of-the-art baselines at both statement and function level by achieving top precision, F1-score, and recall. Only LineVD achieved higher PRAUC at the function level on Big-Vul, but VVulDet$_{noDK}$ is competitive, underscoring a strong and balanced performance. In comparison with transformer-based models, VVulDet$_{noDK}$ achieves performance comparable to RoS-Dex and LineVul, but remains below StagedVulBERT, which demonstrates superior results under its evaluation setting.

### 5.2. RQ2 — VVulDet$_{noDK}$ for different programming languages

We trained and evaluated VVulDet$_{noDK}$ using various programming languages, including C/C++, Java, and Python. We also applied the original LineVD model to each dataset, as a baseline for the comparison. To this aim, we had to modify the code of LineVD to allow the tool to analyze Java and Python code. The results are presented in Table 9, at the statement and function levels.

At statement-level, VVulDet$_{noDK}$ outperforms LineVD for precision, recall, and F1-score (respectively, on average, +0.133, +0.082, and +0.008), while it achieves a lower result for PRAUC (−0.031), showing that the strongly imbalanced nature of the dataset at statement-level can play a crucial role.

At function-level, VVulDet$_{noDK}$ is effective in detecting vulnerabilities across different programming languages, it achieves consistently higher performance compared to the statement-level results for all datasets. For example, for Megavul (Java), the F1-score increases from 0.507 at the statement level to 0.755 at the function level. Similarly, ProjectKB (Java) shows an improvement from 0.586 to 0.760, and even the smallest gain, observed for Big-Vul (C/C++), is notable. Furthermore, at function level, VVulDet$_{noDK}$ outperforms other methods on all datasets (on average for all datasets, +0.060, +0.069, +0.085, and +0.07 respectively for precision, F1-score, recall, and PRAUC). These improvements suggest that aggregating more contextual information (by including DDG edges in the graph construction pipeline ) at the function level enables the model to better capture the relationships among code elements that are relevant for vulnerability detection.

Subsequent observations suggest that VVulDet$_{noDK}$ performs more effectively on Megavul, CVEFixes, and ProjectKB than on Big-Vul. This can be attributed to (i) the complexity of the functions' code, which is higher in C/C++ than in the other languages (e.g., code-specific characters often generate non-unified embeddings compared to text); and (ii) the diversity of the C/C++ code in the BigVul functions tends to be higher than that in other languages.

**RQ2:** The results highlight the VVulDet$_{noDK}$ model's ability to distinguish between vulnerability-related and non-vulnerability-related code elements across different languages, with a particularly strong advantage when applied at the function level.

### 5.3. RQ3 — impact of integrating domain-specific information

To evaluate the effect on the model performance of considering the domain knowledge information while training the model, we compared the results of VVulDet with and without the use of the domain knowledge. We focus on VVulDet, as it outperforms the original LineVD.

We systematically retrain each model five times, considering three possible sources of domain knowledge information: (i) the natural-language CVE descriptions; (ii) the natural-language CWE descriptions; and (iii) the reference code attached as examples to the natural-language CWE descriptions (*RefCode* in Table 10).

As the first form of domain knowledge, we focus on the descriptions of CVEs and CWEs, which are integrated into the detection method by providing the tool as input to the models' training. Specifically, both CWE and CVE textual descriptions are tokenized and transformed into embeddings (i.e., vectors). These domain embeddings are then incorporated into a subgraph as additional node features. During the training, such embeddings are concatenated with the traditional

**Table 9**
Performance metrics of LineVD and VVulDet$_{noDK}$ in different programming languages at both function and statement-level (best in **bold**).

| Dataset | Language | Statement-level | | | | Function-level | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Prec | F1-score | Rec | PRAUC | Prec | F1-score | Rec | PRAUC |
| LineVD | | | | | | | | | |
| ProjectKB | Java | 0.509 | 0.504 | 0.500 | 0.504 | 0.620 | 0.652 | 0.688 | 0.613 |
| Megavul | Java | 0.382 | 0.439 | **0.516** | **0.538** | 0.631 | 0.643 | 0.652 | 0.615 |
| CVEFixes | Python | 0.522 | 0.514 | 0.508 | **0.518** | 0.617 | 0.627 | 0.603 | 0.585 |
| Big-Vul | C/C++ | 0.271 | 0.360 | 0.533 | **0.642** | 0.542 | 0.615 | 0.713 | 0.617 |
| VVulDet$_{noDK}$ | | | | | | | | | |
| ProjectKB | Java | **0.667** | **0.586** | **0.522** | **0.518** | **0.714** | **0.760** | **0.812** | **0.733** |
| Megavul | Java | **0.503** | **0.508** | 0.512 | 0.508 | **0.721** | **0.756** | **0.794** | **0.713** |
| CVEFixes | Python | **0.523** | **0.518** | **0.513** | 0.517 | **0.623** | **0.647** | **0.670** | **0.607** |
| Big-Vul | C/C++ | **0.523** | **0.533** | **0.542** | 0.533 | **0.593** | **0.651** | **0.722** | **0.657** |

**Table 10**
MITRE Reference code — CWEs and code fragments per dataset and programming language.

| Dataset | Language | Reference code | |
|---|---|---|---|
| | | Match CWE | Nr. of code fragments in MITRE |
| ProjectKB<br>MegaVul | Java<br>Java | 22<br>40 | 329 |
| CVEFixes | Python | 9 | 31 |
| BigVul | C/C++ | 91 | 382 |

source code representations (e.g., source code vector) to form enriched feature vectors for each node. This combined feature representation allows the GNN-based vulnerability detection model to jointly learn from both structural/code-level information and domain-specific vulnerability knowledge.

In contrast, an alternative mechanism is employed to assess the influence of the reference code. For this, we computed the cosine similarity between the function under analysis (i.e., the function used to construct the subgraph) and its corresponding reference code, which describes the same CWE. This similarity score is then utilized to weight the nodes and edge features in each subgraph. The cosine similarity coefficient is computed as follows.

Let $A$ and $B$ represent two non-null vectors, where $A$ corresponds to the embedded code function of a sample linked to a specific CWE category, and $B$ represents the embedded code function for the same CWE within a dataset. The cosine similarity, which ranges from −1 to 1, measures the degree of similarity between $A$ and $B$, and it is calculated using the Euclidean dot product formula:

$$AB = \|A\|\|B\|\cos(\Theta), \tag{18}$$

where the cosine similarity $S_C$ is given by:

$$S_C = \cos(\Theta) = \frac{AB}{\|A\|\|B\|}.$$

This simplifies to:

$$S_C = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}. \tag{19}$$

Additionally, the embedded vector of the reference code is introduced as a node feature within the graph, enriching the subgraph with more informative insights regarding vulnerabilities present in the source code. Table 10 (last column) reports the number of reference code fragments we have identified in the MITRE system, for each programming language. Table 10 (column "Match CWE") also reports the number of matched CWEs for which we identified reference code fragments, then used to enrich the datasets.

To evaluate the impact of incorporating domain knowledge, we systematically retrain the model and report the associated evaluation metrics. Fig. 2 summarizes the results obtained at the statement and function levels, presenting metrics for an overall view of improvements. The baseline values are detailed in Table 9 for the **VVulDet$_{noDK}$** model.

At the statement level, considering the average value of the different types of knowledge incorporation in VVulDet$_{noDK}$, we observe an overall improvement in metrics in all datasets for VVulDet. In fact, only recall for BigVul (−0.010) and PRAUC for CVEFixes (−0.004) show a slightly decreasing result when domain knowledge is incorporated. On average, precision (+0.035), F1-score (+0.024), recall (+0.030), and PRAUC (+0.034) improve, suggesting a generally better detection rate. Fig. 2 presents the average of all metrics for each method and across domains, where the performance gains remain consistently smaller than at the function level, however. This can be attributed to the fact that the same high-level CVE/CWE domain descriptions and reference code are applied to both function and statement representations, which may not capture the fine-grained semantics of individual statements as effectively as those of complete functions. Moreover, the severe class imbalance at the statement level often causes the model to achieve deceptively high accuracy while failing to generalize, as reflected by PRAUC values (less than 70%) of the model to address positive cases.

At the function level, by considering the average value of the different types of knowledge incorporation over VVulDet$_{noDK}$, we observe a substantial improvement in the metrics over the datasets for VVulDet (on average, precision +0.105, F1-score +0.098, recall +0.104, and PRAUC +0.067). The pronounced function-level improvements may stem from domain knowledge providing a more explicit and formal description of vulnerable function behaviors than what is available at the statement level, thus enabling the model to better capture function-wide vulnerability patterns.

Regarding the impact of each type of knowledge information incorporated over VVulDet$_{noDK}$, we observe that combining *CVE* and *CWE* textual descriptions typically results in the most consistent and balanced performance gains. This synergy aligns with the complementary nature of CWEs, which describe the underlying causes of vulnerabilities, and CVEs, which document observed vulnerability instances. *RefCode* exhibits positive but limited gains, particularly in recall. This limited improvement likely reflects the relatively sparse coverage of reference code fragments in the MITRE database for the CWEs represented in our datasets (see Table 10). Nonetheless, RefCode provides complementary information that aids detection. By focusing on the datasets, we notice that BigVul at the statement level is the one in which the impact of domain knowledge information is lower, particularly in recall. In this case, the combination of *CVE* and *CWE* is vital to achieve a higher result with respect to the ones of VVulDet$_{noDK}$. This can be due to the high complexity and variety of the C++ code in the BigVul dataset.

The substantial function-level improvement is also attributed to the semantic alignment between domain-specific knowledge and function-level representations. CWE and CVE descriptions, along with MITRE

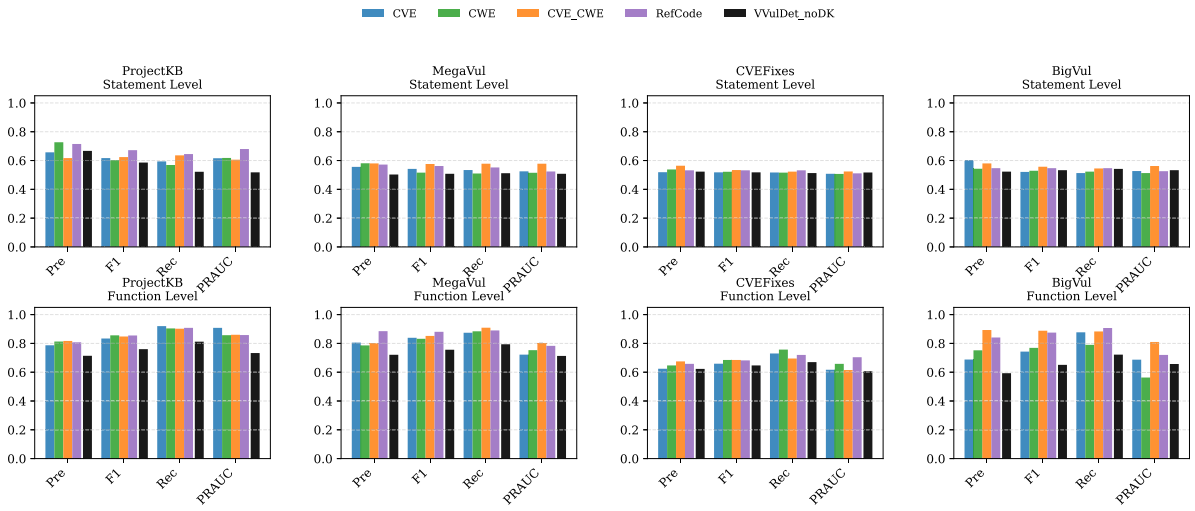



**Fig. 2.** Performance Comparison: Baseline vs Domain Knowledge Methods.

reference implementations, encode vulnerability characteristics in terms of control-flow patterns, data-flow dependencies, and functional misuse scenarios, which are inherently expressed across multiple statements rather than isolated lines of code. As a result, these domain-specific features more explicitly formalize vulnerable function behavior than individual statement representations. In contrast, at the statement level, vulnerability semantics are distributed and indirect, leading to weaker alignment between injected domain knowledge and local node features. Moreover, since GATs propagate information through local neighborhoods (see Section 3.2), globally injected domain description vectors traverse multiple attention layers to influence individual nodes, which may also lead to semantic dilution and reduced effectiveness at the statement level.

**RQ3:** We conclude that the incorporation of the domain knowledge information helps VVulDet in better learning vulnerability patterns, thus improving the performance. The substantial function-level improvement is also because domain-specific features more explicitly formalize vulnerable function behavior compared to statement-level representations, thus enabling the model to better capture vulnerability contexts. While *RefCode* contributes positively (recall mostly), its improvement is somewhat limited, possibly due to the relatively small number of available reference code fragments in MITRE. The combined use of CVE and CWE descriptions tends to yield the most consistent benefits.

### 5.4. RQ4 — VVulDet compared to sequential models incorporating domain knowledge and LLM

To compare the performance of graph-based models and transformer-based models when enhanced with domain knowledge information, we compared the results of VVulDet with those of eSVDet [33], a method designed to incorporate knowledge information into SVDet [34], a JavaBERT-based transformer, and an LLM model that performs vulnerability detection at both function and statement levels. We selected eSVDet since, to the best of our knowledge, it is the unique sequential model that has been enhanced with domain knowledge information. However, the integration of this knowledge differs from our approach due to architectural distinctions. Concerning the LLMs, we included a zero-shot use of *Claude-Sonnet-3* and *Llama 3* from AWS[14] Bedrock, along with the fine-tuned versions of *Llama-3.1-8B-Instruct*, *Mistral-7B-Instruct*. Specifically, we developed a Python script to iteratively query the model for each test sample using the following message prompt:

```
messages = [
{
 "role": "user",
  "content": [{  "text": (
    "You are a software vulnerability detection assistant.\n"
    "Analyze the source code below and reply strictly in JSON with:\n"
    "  - vul_status: 'yes' or 'no'\n"
    "  - vulnerable_lines: list of line(s) or empty if none.\n\n"
    f"Code:\n{source_code}\n\n"
    "Respond only in JSON, without explanations."
    ) }], } ]
```

The model's JSON outputs were compared with the ground-truth labels in the dataset to compute the evaluation metrics. Furthermore, we also included *LineVul* (without domain knowledge), which have been reported in the literature as LLMs capable of analyzing and reasoning about software security [35]. However, we only considered the function-level metrics of LineVul, as its statement-level evaluation, based on top-(k) accuracy, differs from our approach.

eSVDet is based on a different architecture (i.e., transformer) and adopts several mechanisms to incorporate domain knowledge information, only partially comparable with the ones used in this work. Hence, for a fair comparison, we focused on comparing the two methods under the same conditions, with regard to: (i) domain-specific knowledge information; (ii) evaluation metrics; and (iii) dataset. For this reason, we focus on ProjectKB (Java) and on the following domain knowledge incorporation mechanisms used by eSVDet (for more details see: [33]):

- **Multi-label (eSVDet$_M$)**: the model is used as a multi-label classifier, labeling each input code fragment with a composite label that encodes both the vulnerability presence label and associates CWEs via one-hot encoding.
- **CWE hierarchy for the loss function (eSVDet$_H$)**: The CWE hierarchy is used to weight predictions during loss computation, guiding the learning process by considering CWE relationships.
- **Similarity with MITRE reference code for the loss function (eSVDet$_S$)**: It weighs the CWE predictions in the loss function based on the cosine similarity between code fragments and MITRE reference code, emphasizing similarity during learning.
- **Multi-label as input (eSVDet$_{iM}$)**: The multi-label one-hot vector is concatenated with SVDet's classification output and processed by a two-layer MLP to learn non-linear combinations before the final classification.

[14] https://aws.amazon.com/

- **CWE hierarchy as input** (**eSVDet**$_{iH}$): Similar to eSVDet$_H$, but the weighted CWE vector is fed as input to the MLP rather than used only in the loss.

Tables 11 and 12 report the results collected at both statement and function levels, respectively, without and with domain knowledge. The results show that the sequential model also benefits from domain knowledge, outperforming the well-known LLMs Claude and Llama on the simple zero-shot use, particularly at the statement level, even though the PRAUC value indicates sensitivity to the imbalanced nature of the dataset (on average across the different incorporation mechanisms: precision +0.113, F1-score +0.135, recall +0.145, and PRAUC –0.2).

On the other hand, while most versions of the transformer-based SVDet achieve nearly comparable metrics, such as F1 and recall, with and without domain information, the LineVul transformer model achieves the highest precision compared to other models, including those that incorporate domain knowledge. In the LLM-based models, a distinct advantage is observed when tuning the model with training data before evaluation with the test set, particularly in recall gains, reaching 0.725 for *Llama-3.1-8B-Instruct* and 0.619 for Mistral-7B-Instruct, which outperform SVDet in recall. However, this improvement does not extend to overall metrics, as a significant decrease in precision and F1-score is noted for these models. This raises questions about the sufficiency of the training samples, as these models possess a greater number of parameters than the data can adequately support.

The different domain knowledge incorporation mechanisms yield broadly comparable results within each granularity level. In particular, only eSVDet$_S$ exhibits a markedly different trend, performing better at statement-level than at function-level, probably due to the limited availability of reference code fragments for the CWEs considered.

Overall, VVulDet outperforms all eSVDet variants and the LLMs at both the statement level (averaged across different incorporation mechanisms: precision +0.301, F1-score +0.267, recall +0.173, and PRAUC +0.079) and the function level (averaged across different incorporation mechanisms: precision +0.147, F1-score +0.278, recall +0.297, and PRAUC +0.298). However, VVulDet does not surpass LineVul in precision. Furthermore, the improvements in evaluation metrics resulting from the incorporation of domain knowledge are more pronounced for VVulDet at the function level and for eSVDet at the statement level.

**RQ4:** The GNN-based model excels at both the statement and function levels of vulnerability detection, likely due to the GNNs' ability to capture the structural context of code. While domain information enhanced the function-level analysis in GNN variants, the statement-level transformer-based model benefited more compared to the function-level vulnerability analysis. Moreover, this indicates a need for additional investigation into optimizing the training of these models, particularly regarding the potential incorporation of domain information.

## 6. Discussion

We proposed an enhanced GNN-based approach for vulnerability detection by extending: (i) the code representation with Data Dependency Graphs (DDG) to model the flow of values from variable definitions to uses; (ii) the embedding mechanism with Node2vec to better capture long-term dependencies between code elements; and (iii) the learning process with domain-specific knowledge to complement the structural information from source code.

Experimental results across multiple datasets and programming languages show that the integration of domain knowledge improves detection performance, particularly at the function level, where VVulDet constantly outperforms all baselines in terms of precision, F1-score, and recall, e.g., VVulDet achieves, on average, up to +30% precision and +19.5% recall. A notable gain is also achieved by VVulDet at statement level as well: on average, it improves up to +7.8% precision and +6.6% recall. Compared to the original LineVD results, we observe that VVulDet strongly outperforms its original version at both statement and function level, respectively up to +32.9% precision and +14.0% recall for statement level, and +35.1% precision and +20.4% recall for function level. In terms of absolute values, the highest performance of VVulDet is achieved in the Java datasets for which, on average, at function level the model attains precision and recall rates as high as 80.6% and 91.1%, while 67.9% and 61.1% at statement level. These results highlight that domain knowledge helps the GNN model better generalize beyond purely structural cues, especially for less frequent vulnerable cases. Improving both precision and recall is critical in vulnerability detection, as it allows the model to more reliably identify true vulnerabilities while reducing false positives, regardless of dataset imbalance.

Furthermore, we conducted a fine-grained analysis by aggregating the reported metrics across all datasets and grouping results by Common Weakness identifiers. Fig. 3 presents the averaged performance for the top-10 most frequent CWEs. The results show that the impact of domain knowledge varies substantially across vulnerability categories. In particular, CWE-502 ("Deserialization of Untrusted Data") and CWE-20 (Improper Input Validation), categories related to system's input validation, as well as CWE-611 ("Improper Restriction of XML External

**Table 11**
Vulnerability detection without domain knowledge on the ProjectKB dataset.

| Model | Prec | F1-score | Rec | PRAUC |
|---|---|---|---|---|
| Statement-level | | | | |
| VVulDet$_{noDK}$ | **0.667** | **0.586** | **0.522** | 0.518 |
| SVDet | 0.306 | 0.234 | 0.289 | **0.770** |
| Claude$_{LLM}$ | 0.408 | 0.380 | 0.484 | 0.413 |
| Llama$_{LLM}$ | 0.388 | 0.384 | 0.470 | 0.392 |
| LLama$_{Tune}$ | 0.035 | 0.065 | **0.512** | 0.035 |
| Mistral$_{Tune}$ | 0.249 | 0.262 | 0.276 | 0.256 |
| Function-level | | | | |
| VVulDet$_{noDK}$ | 0.714 | 0.760 | **0.812** | **0.733** |
| SVDet | 0.762 | **0.775** | 0.799 | 0.280 |
| LineVul | **0.859** | 0.746 | 0.659 | 0.713 |
| Claude$_{LLM}$ | 0.574 | 0.535 | 0.531 | 0.582 |
| Llama$_{LLM}$ | 0.565 | 0.542 | 0.536 | 0.565 |
| LLama$_{Tune}$ | 0.122 | 0.2089 | **0.725** | 0.122 |
| Mistral$_{Tune}$ | 0.323 | 0.425 | 0.619 | 0.385 |

**Table 12**
Vulnerability detection with domain knowledge on the ProjectKB dataset.

| Model | Prec | F1-score | Rec | PRAUC |
|---|---|---|---|---|
| Statement-level | | | | |
| VVulDet$_{CWE}$ | 0.727 | 0.602 | 0.569 | 0.618 |
| VVulDet$_{RefCode}$ | **0.715** | **0.672** | **0.645** | 0.680 |
| avg | 0.721 | 0.637 | 0.607 | 0.649 |
| eSVDet$_M$ | 0.391 | 0.354 | 0.423 | 0.630 |
| eSVDet$_{iM}$ | 0.396 | 0.350 | 0.408 | 0.530 |
| eSVDet$_H$ | 0.399 | 0.311 | 0.334 | 0.710 |
| eSVDet$_{iH}$ | 0.406 | 0.348 | 0.460 | 0.570 |
| eSVDet$_S$ | 0.504 | 0.485 | 0.545 | 0.410 |
| avg | 0.4192 | 0.3696 | 0.434 | 0.57 |
| Function-level | | | | |
| VVulDet$_{CWE}$ | 0.813 | 0.856 | 0.904 | 0.857 |
| VVulDet$_{RefCode}$ | **0.808** | **0.855** | **0.918** | **0.858** |
| avg | 0.8105 | 0.8555 | 0.911 | 0.8575 |
| eSVDet$_M$ | 0.793 | 0.694 | 0.676 | 0.710 |
| eSVDet$_{iM}$ | 0.802 | 0.650 | 0.620 | 0.700 |
| eSVDet$_H$ | 0.794 | 0.686 | 0.660 | 0.650 |
| eSVDet$_{iH}$ | 0.804 | 0.660 | 0.620 | 0.610 |
| eSVDet$_S$ | 0.121 | 0.194 | 0.491 | 0.124 |
| avg | 0.6628 | 0.5768 | 0.6134 | 0.5588 |

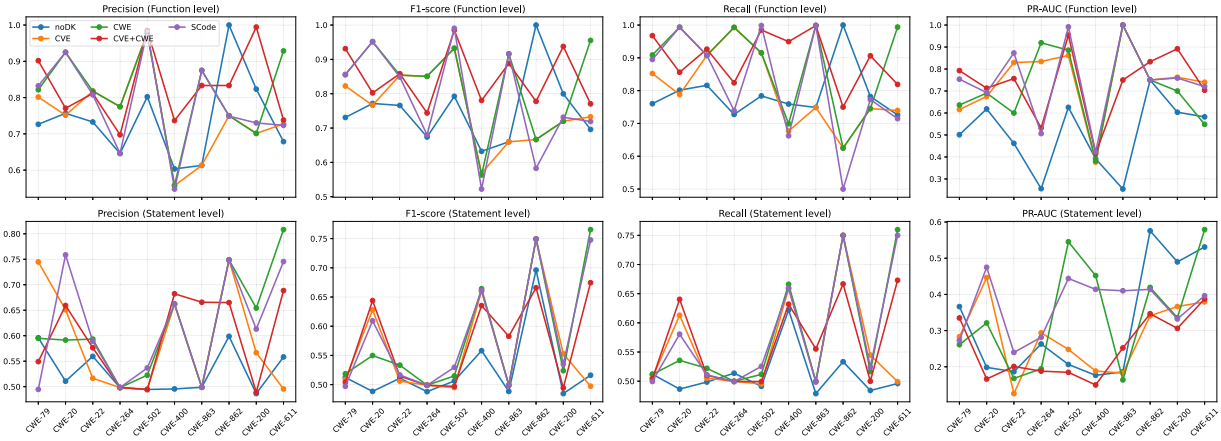



**Fig. 3.** Performance of Top 10 CWEs with and without Domain Knowledge.

Entity Reference"), category related to system resources access control and restricting, show notable improvements in precision and recall when domain information is incorporated. On the other hand, CWE-862 ("Missing Authorization"), category related to mechanisms for user's authorization, shows only marginal gains at the statement level in terms of F1-score, while achieving optimal function-level performance even without domain knowledge. Thus, certain vulnerability types are sufficiently captured by structural and data-dependency cues, whereas others benefit more from enriched semantic context.

In contrast, transformer-based baselines generally perform better at the function level than at the statement level, benefiting from domain-specific tokens and broader semantic context. However, they still underperform compared to GNNs for fine-grained, statement-level vulnerability detection, reflecting GNNs' strength in modeling detailed structural relationships within code. In transformers, statement-level predictions rely on weighted token contributions, which may limit their ability to capture precise structural dependencies. Like VVulDet, transformer variants outperform the LLMs in identifying vulnerable code, although this advantage may partly stem from the LLMs not being fine-tuned with domain knowledge. This suggests that relying solely on LLMs for vulnerability detection may be insufficient without additional domain knowledge, a point we plan to address in future work.

> Injecting domain knowledge into graph-based vulnerability detection significantly improves both statement and function-level performance, with the largest gains at the function level and consistent, measurable improvements at the statement level. The lower performance of LLMs highlights the need for further optimization, such as domain refinement in vulnerability identification.

### 6.1. Computational requirements

Graph-based models excel at both coarse and fine-grained tasks like function and statement-level vulnerability detection by capturing detailed code semantics and interdependencies. However, their practical use on large-scale codebases is hindered by heavy computational demands and reliance on external tools (e.g., Joern, PHPParser) to extract complex structural graph-based representations such as ASTs, CFGs, and CPGs. The processing phase for extracting such representations alone can be extremely time-consuming, taking up to several days for datasets like BigVul and ProjectKB.

In contrast, sequence-based models work directly on raw or tokenized source code, avoiding costly structural analysis. This result is a more efficient preprocessing and faster execution. Moreover, sequential models are particularly effective for multi-level classification tasks, including function-level and statement-level detection, where GNNs may struggle due to scalability and architectural constraints.

Consequently, the choice between graph-based and sequence-based models can depend on the task granularity and resource availability: graph models offer superior statement-level precision, while sequence models provide greater efficiency and flexibility for broader, multi-level vulnerability detection. Balancing computational cost with detection performance is key to selecting the appropriate approach for each use case.

### 6.2. Threats to validity

*Internal validity.* concerned with extraneous factors that can influence results. One threat can be related to the datasets. Different datasets can lead to different results, so we selected well-known datasets composed of real code functions from open-source projects, and that contain all the information needed to apply the proposed approach. Additionally, inconsistencies, noisy information, and wrong labels of code in a dataset could affect the training and evaluation of models. For limiting the impact of such issues, we applied a preprocessing step of data cleaning. Another threat can concern the downsampling of non-vulnerable functions adopted to address the dataset's class imbalance. While this mitigates bias towards non-vulnerable instances, it may also remove important contextual information from the dataset, possibly affecting the learning process.

*External validity.* concerns with factors that can limit the generalizability of the achieved results. Although we evaluated our method on datasets covering different programming languages, the result may not generalize to other languages where domain information is not widely studied. Moreover, our evaluation setting, although rigorous, may not reflect operational deployment scenarios, where code evolves over time, vulnerabilities may be hidden, and ground-truth labels not available.

*Construct validity.* concerns with factors that can influence the operationalizations of the study. Existing datasets can be curated for academic research, thus they may not fully represent the diversity and complexity of real-world industrial code. We hence selected large datasets composed of a high number of real code functions, and that have been already used in several studies. One threat can be related to the used GNN approach; a different architecture can lead to different results. Hence started from an existing approach and tried to improve it by incorporating the domain knowledge information. Furthermore, we acknowledge that domain sources richer than the MITRE references, which provide extra domain information such as commit messages, issue trackers, and user comments, more vulnerable code examples, and contextual semantic information, were not utilized in this study. However, these sources may contain relevant security indicators, and we will consider incorporating them in our future experimentation.

*Conclusion validity.* concerns whether the observed effect is due to the tool/treatment or if it can be influenced by other factors. While we used standard metrics for reporting the performance, the high statement-level class imbalance of datasets may inflate or obscure these metrics. Moreover, size and complexity of code could have increased the performance variance across different datasets' splits and runs. To limit this threat, we adopted datasets of real open-source projects. However, only further repetitions of the study can corroborate the achieved results. We also acknowledge the use of a single fusion methodology, specifically concatenation. While cross-attention-based fusion presents a viable alternative, it incurs significantly higher computational costs. We opted for the presented method since existing literature on multimodal learning indicates that when auxiliary modalities are global descriptors (such as in our case the used domain knowledge information), linear fusion is often preferable, as it provides an effective and robust integration strategy while avoiding the additional complexity and computational overhead of attention-based mechanisms [36–38]. However, the adoption of more advanced fusion mechanisms can be object of an future investigations.

## 7. Related work

Recent advancements in vulnerability detection have transitioned from static code analysis techniques to AI-driven approaches [16]. Increasing attention has been directed toward AI-driven models for identifying vulnerabilities in source code, which can generally be divided into sequence-based and graph-based methodologies [8].

For example, VulDeePecker [39] encodes code segments into fixed-length symbol vectors, which are subsequently analyzed by using a Bidirectional Long Short-Term Memory (BLSTM) network. Similarly, transformer-based models have been proposed to address vulnerability detection in different programming contexts: [40] developed VDet, while [9] introduced LineVul for identifying vulnerabilities in Java functions and C/C++ statements, respectively. Despite significant advancements in sequence-based models, recent research highlights their persistent challenges, such as the inability to work with highly structured code. Recent models, such as StagedVulBERT [29] and RoS-Dex [30], employ novel transformer designs. Although RoS-Dex is an ensemble-based approach that uses local rotary positional embeddings to enhance positional information and preserve the structural layout of the source code, it did not outperform StagedVulBERT on the BigVul dataset. In contrast, StagedVulBERT constructs source code embeddings through independent representation of code sequences, merging these embeddings to account for the entire function. Yuan J. et al. [41] employed the StagedVulBERT code representation module to develop a detection framework based on reinforcement learning. While this process enables the embedding representation of long code snippets, it insufficiently preserves contextual relationships. Subsequently, Ahmed et al. [42] introduced a detection framework that integrates slice-based CPG with large language models. Although this approach demonstrates robustness against code transformations, the slice CPG component may inconsistently omit useful patterns in the vulnerability detection workflow. Specifically, the model does not support low-granularity detection, such as at the line level. Furthermore, it exhibits inconsistent accuracy when handling highly complex inter-functional relationships. Jiayuan et al. [43] employed contrastive learning to generate precise representations of vulnerable code under the supervision of natural language descriptions. However, their method does not address vulnerability localization.

Graph-based approaches leverage graphical representations of code, such as program dependency graphs and abstract syntax trees, to capture syntactic, semantic, and structural features. Devign [32] represents the source code by means of an AST-enriched graph representation, then feeds it into a Gated Graph Recurrent Layer and a convolutional neural network to embed such a representation and predict the code vulnerability. FUNDED [44], IVDetect [23], LineVD [11], VulGraB [45], and DGVD [46] are more recent techniques that utilize advanced learning architectures, such as feature-attention graph convolutional networks and gated graph neural networks. LineVD and DetectVul [14] work at statement level and are tailored to C/C++ and Python. MVulD [31] applies a graph-neural network and transformer for a multi-modal analysis of information extracted from the source code. These techniques utilize the structural properties, but their focus has largely excluded Java as a target programming language, leaving a gap in this area. To fill the gap, [47] introduced a framework for detecting function-level vulnerabilities in Java by integrating quantum layers.

Existing methods only depend on features extracted directly from source code, and by training models on historical data, have demonstrated significant effectiveness in vulnerability detection. However, there remains an open question regarding the sufficiency of these models in acquiring comprehensive vulnerability-related knowledge from the source code under analysis. To address this gap, some research has highlighted the advantages of incorporating additional features to enhance the model's understanding of code.

For instance, providing deep learning models with supplementary contextual information about the code being analyzed can yield improved performance [48]. [49] emphasizes the role of commit messages in enhancing security path detection, employing a transformer-based architecture to achieve better results. Similarly, [50] explored the utility of source code comments, leveraging models like CodeBERT and RoBERTa to capture semantic and syntactic structures of the code, while incorporating the Doc2vec model in the system for feature embedding. Other notable efforts include [51], who applied cross-domain adaptation techniques to address domain-specific challenges that may affect detection accuracy. [52] proposes a transformer-based method to map natural language CVE descriptions to CWEs, while [53] develops a hybrid approach combining local code snippet information with global dataset-level insights to improve detection outcomes.

The integration of domain-specific knowledge, such as CWE-related data, has also emerged as a promising strategy. [33] demonstrates how incorporating CWE data can boost the performance of transformer-based models. Similarly, [54] introduces an approach for predicting security bug reports using a knowledge graph constructed with CWE and CVE information. Additional study by [55] explored the inclusion of data sources such as commit histories to enhance model performance further. [56] introduces the use of CWE descriptions and reference code to construct a knowledge graph then used to identify the CWE that makes a code fragment vulnerable.

Unlike previous work that focuses on integrating auxiliary information like comments and commit histories, this research emphasizes the incorporation of reference function cosine similarity, CWE, and CVE-related data. Moreover, in contrast to the rule-based entity recognition method used by [54], which relies on knowledge graphs for predicting security bug reports, this study aims to develop an automated approach that directly incorporates CWE, CVE knowledge, and vulnerable reference functions into vulnerability detection models for analyzing source code.

## 8. Conclusion

We presented VVulDet, a GNN model designed for vulnerability detection in source code. Unlike previous GNN architectures, that relied solely on source code, VVulDet employs novel mechanisms to enhance this process by introducing new types of edge on the graph and developing new methods to incorporate additional data sources into the model learning process. These sources include CVE and CWE descriptions, as well as CWEs' reference code samples. The experimentation conducted across diverse datasets has demonstrated the effectiveness of domain knowledge integration mechanisms in the GNN learning process.

Future work will explore the integration of knowledge from other relevant sources, such as code comments, commit messages, and other pertinent information related to vulnerabilities.

**Table 13**
Detection performance with different sources of incorporated Domain Knowledge at the statement and function levels.

| Method | Statement-Level | | | | Function-Level | | | |
|---|---|---|---|---|---|---|---|---|
| | Prec | F1-score | Rec | PRAUC | Prec | F1-score | Rec | PRAUC |
| | ProjectKB | | | | | | | |
| CVE | 0.657 | 0.617 | 0.594 | 0.616 | 0.787 | 0.834 | **0.920** | **0.908** |
| CWE | **0.727** | 0.602 | 0.569 | 0.618 | 0.813 | **0.856** | 0.904 | 0.857 |
| CVE_CWE | 0.617 | 0.624 | 0.636 | 0.606 | **0.817** | 0.848 | 0.902 | 0.860 |
| RefCode | 0.715 | **0.672** | **0.645** | **0.680** | 0.808 | 0.855 | 0.918 | 0.858 |
| avg | 0.679 | 0.629 | 0.611 | 0.630 | 0.806 | 0.848 | 0.911 | 0.871 |
| | Megavul | | | | | | | |
| CVE | 0.556 | 0.542 | 0.534 | 0.525 | 0.806 | 0.839 | 0.874 | 0.722 |
| CWE | **0.581** | 0.516 | 0.510 | 0.515 | 0.786 | 0.832 | 0.884 | 0.753 |
| CVE_CWE | 0.580 | **0.576** | **0.578** | **0.578** | 0.802 | 0.852 | 0.909 | **0.803** |
| RefCode | 0.572 | 0.562 | 0.552 | 0.524 | **0.885** | **0.915** | **0.950** | 0.783 |
| avg | 0.572 | 0.549 | 0.544 | 0.536 | 0.820 | 0.860 | 0.904 | 0.765 |
| | CVEFixes | | | | | | | |
| CVE | 0.519 | 0.518 | 0.517 | 0.508 | 0.624 | 0.659 | 0.730 | 0.617 |
| CWE | 0.538 | 0.522 | 0.516 | 0.507 | 0.647 | 0.685 | **0.757** | 0.658 |
| CVE_CWE | **0.564** | **0.534** | 0.523 | **0.524** | **0.675** | 0.685 | 0.695 | 0.615 |
| RefCode | 0.532 | 0.532 | **0.532** | 0.511 | 0.658 | **0.692** | 0.750 | **0.704** |
| avg | 0.538 | 0.527 | 0.522 | 0.513 | 0.651 | 0.680 | 0.733 | 0.648 |
| | Bigvul | | | | | | | |
| CVE | **0.600** | 0.521 | 0.513 | 0.527 | 0.688 | 0.743 | 0.877 | 0.687 |
| CWE | 0.543 | 0.529 | 0.523 | 0.513 | 0.752 | 0.769 | 0.790 | 0.563 |
| CVE_CWE | 0.580 | **0.557** | 0.545 | **0.562** | **0.893** | **0.888** | 0.883 | **0.810** |
| RefCode | 0.547 | 0.547 | **0.547** | 0.526 | 0.841 | 0.875 | **0.917** | 0.720 |
| avg | 0.568 | 0.539 | 0.532 | 0.532 | 0.794 | 0.819 | 0.867 | 0.695 |

## CRediT authorship contribution statement

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

## Appendix A. Detailed metrics of domain knowledge advancements

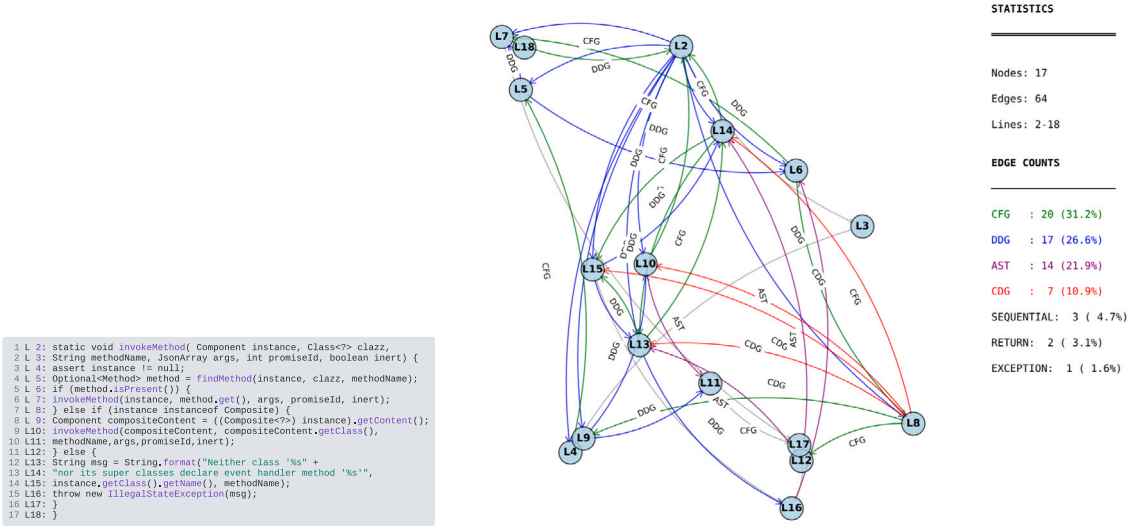
In Table 13, we present the metrics summarized in Fig. 2, providing a detailed comparison of the results that highlight key findings and trends observed in our analysis of domain knowledge.

## Appendix B. Case study

In this case study, we randomly select a vulnerable Java code snippet associated with CWE-200 (“Exposure of Sensitive Information to an Unauthorized Actor”) from the MegaVul dataset. This sample is incorrectly classified as non-vulnerable by the LLM-based approaches considered in our experiment, but is correctly identified as vulnerable by both JLineVD (i.e., our baseline graph-based model that corresponds to the improved version of LineVD as described by [12]) and our proposed model **VVulDet**, which incorporates data-dependence edges and domain knowledge derived from CWE and CVE descriptions. Fig. 4 presents the analyzed example. Specifically, Fig. 4(a) shows the source code method, while Fig. 4(b) illustrates the corresponding Code Property Graph extracted using *Joern*. In the graph representation, major edge types are represented using distinct colors, and the distribution of edge types is reported for clarity. Node identifiers are denoted as $L_x$ to facilitate reference between the source code and the graph representation.

Fig. 5 compares the line-level vulnerability predictions produced by JLineVD and **VVulDet**. For each subfigure, the first column represents the node-level ground truth, obtained by aligning vulnerable code versions as described in Section 4.2. The second column shows binary predictions at the statement level using a vulnerability threshold of 0.5, the third column shows the line numbers, while the heatmap intensity reflects the predicted vulnerability probabilities.

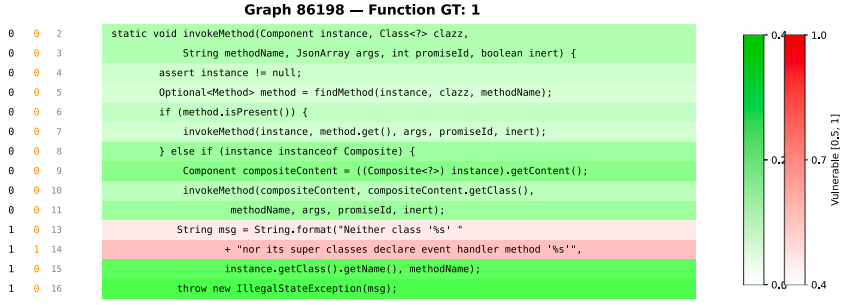
We can observe that although JLineVD correctly classifies the function as vulnerable, it identifies only one true vulnerable line with high confidence. In contrast, **VVulDet** not only detects the vulnerable function but also correctly highlights three vulnerable code lines with higher prediction probabilities. While VVulDet additionally assigns elevated vulnerability scores to lines 6, 8, 9, and 11, which are not labeled as ground truth vulnerabilities; these lines share direct or indirect DDG edges with the true vulnerable statements, as shown in Fig. 4(b). From a practical perspective, such conservative predictions are desirable in vulnerability detection, as they help direct developer attention toward semantically related code regions rather than overlooking potentially risky statements. Since false negatives pose a higher risk than false positives in security-critical contexts, the ability of VVulDet to surface dependency-related code lines demonstrates its effectiveness in supporting secure code review and vulnerability triage. These observations show the potential of the use of the DDG edges and the domain knowledge in source code vulnerability identification with a Graph neural network.



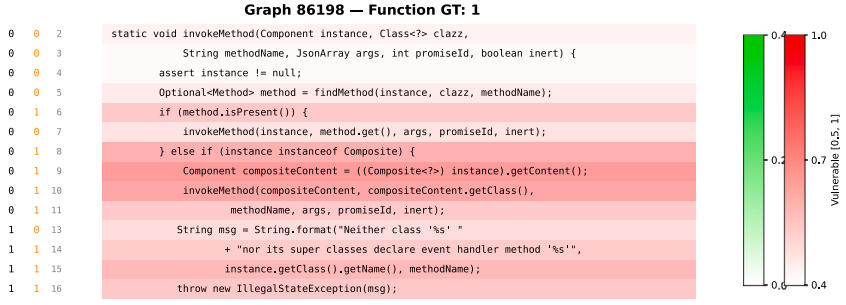

(a) Vulnerable Java code snippet (CWE-200).

(b) Full corresponding Code Property Graph extracted with Joern.

**Fig. 4.** Input representations used in the case study.




(a) Probability predictions without domain knowledge.




(b) VVulDet predictions with DDG and domain knowledge.

**Fig. 5.** Line-level vulnerability heatmaps for the selected CWE-200 example. The green lines represent non-vulnerable predictions with a probability between $[0.0, 0.4]$, while the red lines indicate a range from warning to vulnerable $[0.4, 1.0]$ (vulnerable if $p \geq 0.5$). The color intensity is computed based on prediction confidence; therefore, lines with higher predictions appear more intensified.

## Data availability

The analysis in this study uses well-known public datasets from various sources. Our source code, along with further detailed instructions, is available at https://github.com/SCVDetect/VVulDet/.

## References

[1] M. Felderer, M. Büchler, M. Johns, A.D. Brucker, R. Breu, A. Pretschner, Security testing: A survey, in: Advances in Computers, vol. 101, Elsevier, 2016, pp. 1–51.

[2] F. Mateo Tudela, J.-R. Bermejo Higuera, J. Bermejo Higuera, J.-A. Sicilia Montalvo, M.I. Argyros, On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications, Appl. Sci. 10 (24) (2020) 9119.

[3] K. Goseva-Popstojanova, A. Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Inf. Softw. Technol. 68 (2015) 18–33.

[4] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, J. Hu, Vulpecker: an automated vulnerability detection system based on code similarity analysis, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 201–213.

[5] X. Chen, Q. Li, Z. Yang, Y. Liu, S. Shi, C. Xie, W. Wen, VulChecker: achieving more effective taint analysis by identifying sanitizers automatically, in: 2021

IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom, IEEE, 2021, pp. 774–782.
[6] P. Zeng, G. Lin, L. Pan, Y. Tai, J. Zhang, Software vulnerability analysis and discovery using deep learning techniques: A survey, IEEE Access 8 (2020) 197158–197172.
[7] J. Wang, M. Huang, Y. Nie, J. Li, Static analysis of source code vulnerability using machine learning techniques: A survey, in: 2021 4th International Conference on Artificial Intelligence and Big Data, ICAIBD, IEEE, 2021, pp. 76–86.
[8] Y. Yang, G. Li, On the code vulnerability detection based on deep learning: A comparative study, IEEE Access (2024).
[9] M. Fu, C. Tantithamthavorn, Linevul: A transformer-based line-level vulnerability prediction, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 608–620.
[10] C. Thapa, S.I. Jang, M.E. Ahmed, S. Camtepe, J. Pieprzyk, S. Nepal, Transformer-based language models for software vulnerability detection, in: Proceedings of the 38th Annual Computer Security Applications Conference, 2022, pp. 481–496.
[11] D. Hin, A. Kan, H. Chen, M.A. Babar, Linevd: statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 596–607.
[12] R.Z. Lekeufack Foulefack, A. Marchetto, Enhanced graph neural networks for vulnerability detection in java via advanced subgraph construction, in: Testing Software and Systems, Springer Nature Switzerland, Cham, 2025, pp. 131–148.
[13] A. Grover, J. Leskovec, Node2vec: Scalable feature learning for networks, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, USA, 2016, pp. 855–864.
[14] H.-C. Tran, A.-D. Tran, K.-H. Le, DetectVul: A statement-level code vulnerability detection for Python, Future Gener. Comput. Syst. 163 (2025) 107504.
[15] B. Wu, F. Zou, Code vulnerability detection based on deep sequence and graph models: A survey, Secur. Commun. Networks 2022 (1) (2022) 1176898.
[16] N. Shiri Harzevili, A. Boaye Belle, J. Wang, S. Wang, Z.M. Jiang, N. Nagappan, A systematic literature review on automated software vulnerability detection using machine learning, ACM Comput. Surv. 57 (3) (2024) 1–36.
[17] A. Shihab, M. Alanezi, Exploring datasets in software vulnerability analysis: A review, in: 2024 4th International Conference on Emerging Smart Technologies and Applications, ESmarTA, IEEE, 2024, pp. 1–6.
[18] R.Z. Lekeufack Foulefack, A. Marchetto, Incorporating domain knowledge into GNNs for advanced vulnerability detection in java, in: 2025 IEEE/ACM International Conference on Automation of Software Test, AST, IEEE, 2025, pp. 160–169.
[19] M.J. Haber, B. Hibbert, Asset Attack Vectors: Building Effective Vulnerability Management Strategies to Protect Organizations, A Press, 2018.
[20] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, Y. Liu, Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search, IEEE Trans. Softw. Eng. 49 (4) (2023) 2839–2855.
[21] H.V. Nguyen, J. Zheng, A. Inomata, T. Uehara, Code aggregate graph: Effective representation for graph neural networks to detect vulnerable code, IEEE Access 10 (2022) 123786–123800.
[22] X. Ji, L. Liu, J. Zhu, Code clone detection with hierarchical attentive graph embedding, Int. J. Softw. Eng. Knowl. Eng. 31 (06) (2021) 837–861.
[23] Y. Li, S. Wang, T.N. Nguyen, 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, 2021, pp. 292–303.
[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[25] S.E. Ponta, H. Plate, A. Sabetta, M. Bezzi, C. Dangremont, A manually-curated dataset of fixes to vulnerabilities of open-source software, in: Proceedings of the 16th International Conference on Mining Software Repositories, 2019.
[26] C. Ni, L. Shen, X. Yang, Y. Zhu, S. Wang, MegaVul: AC/C++ vulnerability dataset with comprehensive code representations, in: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR, IEEE, 2024, pp. 738–742.
[27] J. Fan, Y. Li, S. Wang, T.N. Nguyen, A c/c++ code vulnerability dataset with code changes and cve summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.
[28] G. Bhandari, A. Naseer, L. Moonen, Cvefixes: automated collection of vulnerabilities and their fixes from open-source software, in: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30–39.
[29] Y. Jiang, Y. Zhang, X. Su, C. Treude, T. Wang, Stagedvulbert: Multi-granular vulnerability detection with a novel pre-trained code model, IEEE Trans. Softw. Eng. (2024).
[30] C. Do Xuan, D.B. Quang, V.D. Quang, A novel approach for software vulnerability detection based on ensemble learning model, Comput. Electr. Eng. 130 (2026) 110848.
[31] C. Ni, X. Guo, Y. Zhu, X. Xu, X. Yang, Function-level vulnerability detection through fusing multi-modal knowledge, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2023, pp. 1911–1918, http://dx.doi.org/10.1109/ASE56229.2023.00084.
[32] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: Proceedings of the 33rd International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, 2019.
[33] A. Marchetto, R.Z. Lekeufack Foulefack, Towards the use of domain knowledge to enhance transformer-based vulnerability detection, in: A. Bertolino, J.a. Pascoal Faria, P. Lago, L. Semini (Eds.), Quality of Information and Communications Technology, Springer Nature Switzerland, Cham, 2024, pp. 373–390.
[34] A. Marchetto, Can explainability and deep-learning be used for localizing vulnerabilities in source code? in: Proceedings of 5th ACM/IEEE International Conference on Automation of Software Test, AST 2024, ACM, USA, 2024, pp. 110–119.
[35] X. Yin, C. Ni, S. Wang, Multitask-based evaluation of open-source llm on software vulnerability, IEEE Trans. Softw. Eng. (2024).
[36] T. Baltrušaitis, C. Ahuja, L.-P. Morency, Multimodal machine learning: A survey and taxonomy, IEEE Trans. Pattern Anal. Mach. Intell. 41 (2) (2018) 423–443.
[37] J. Gao, P. Li, Z. Chen, J. Zhang, A survey on deep learning for multimodal data fusion, Neural Comput. 32 (5) (2020) 829–864.
[38] B. Yuan, Y. Lu, Y. Fang, Y. Wu, D. Zou, Z. Li, Z. Li, H. Jin, Enhancing deep learning-based vulnerability detection by building behavior graph model, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 2262–2274.
[39] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: Network and Distributed Systems Security (NDSS) Symposium, 2018.
[40] C. Mamede, E. Pinconschi, R. Abreu, A transformer-based IDE plugin for vulnerability detection, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–4.
[41] Y. Jiang, Z. Qu, C. Treude, X. Su, T. Wang, Enhancing fine-grained vulnerability detection with reinforcement learning, IEEE Trans. Softw. Eng. (2025).
[42] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, I. Khalil, $\{LLMxCPG\}$:$\{Context-Aware\}$ vulnerability detection through code property $\{Graph-Guided\}$ large language models, in: 34th USENIX Security Symposium, USENIX Security 25, 2025, pp. 489–507.
[43] J. Li, L. Cui, S. Zhao, Y. Yang, L. Li, H. Zhu, Clever: Multi-modal contrastive learning for vulnerability code representation, in: Findings of the Association for Computational Linguistics, ACL 2025, 2025, pp. 7940–7951.
[44] H. Wang, G. Ye, Z. Tang, S.H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Trans. Inf. Forensics Secur. 16 (2020) 1943–1958.
[45] S. Wang, C. Huang, D. Yu, X. Chen, VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network, Softw.: Pr. Exp. 53 (8) (2023) 1631–1658.
[46] M. Ling, M. Tang, D. Bian, S. Lv, Q. Tang, A dual graph neural networks model using sequence embedding as graph nodes for vulnerability detection, Inf. Softw. Technol. 177 (2025) 107581.
[47] S. Hussain, M. Nadeem, J. Baber, M. Hamdi, A. Rajab, M.S. Al Reshan, A. Shaikh, Vulnerability detection in java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction, Sci. Rep. 14 (1) (2024) 7406.
[48] F. Tian, C. Treude, Adding context to source code representations for deep learning, in: 2022 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2022, pp. 374–378.
[49] F. Zuo, X. Zhang, Y. Song, J. Rhee, J. Fu, Commit message can help: security patch detection in open source software via transformer, in: 2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications, SERA, IEEE, 2023, pp. 345–351.
[50] A. Briciu, M. Lupea, G. Czibula, I.G. Czibula, Enriching the semantic representation of the source code with natural language-based features from comments for improving the performance of software defect prediction, in: ENASE, 2024, pp. 132–143.
[51] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, Y. Xiang, CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation, IEEE Trans. Dependable Secur. Comput. 19 (1) (2020) 438–451.
[52] S.S. Das, E. Serra, M. Halappanavar, A. Pothen, E. Al-Shaer, V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities, in: 2021 IEEE 8th International Conference on Data Science and Advanced Analytics, DSAA, IEEE, 2021, pp. 1–12.
[53] Z. Lu, P. Du, J.-Y. Nie, VGCN-bert: augmenting BERT with graph embedding for text classification, in: Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part I 42, Springer, 2020, pp. 369–382.
[54] W. Zheng, J. Cheng, X. Wu, R. Sun, X. Wang, X. Sun, Domain knowledge-based security bug reports prediction, Knowl.-Based Syst. 241 (2022) 108293.
[55] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.
[56] M. Vecellio Reane, D. Dall'Anese, R.Z. Foulefack, A. Marchetto, Towards a knowledge graph based approach for vulnerable code weaknesses identification, in: IFIP International Conference on Testing Software and Systems, Springer, 2024, pp. 159–166.