



SemVul: Semantic-enhanced graph neural networks for code property graph-based vulnerability detection[☆]

Affan Younas^{a, ID}, Leonardo Regano^{a, ID, *}, Giorgio Giacinto^{a, b, ID}

^a Dipartimento di Ingegneria Elettrica e Elettronica, Università di Cagliari, Via Marengo 3, 09123, Cagliari, Italy

^b Interuniversity National Consortium for Informatics, CINI, Via Ariosto, 25, Rome, 00185, Italy

ARTICLE INFO

Dataset link: <https://github.com/Affan47/SemVul>

Keywords:

Software vulnerability
Vulnerability detection
Graph neural networks
Code property graph

ABSTRACT

Vulnerabilities in source code are often the root cause of cyberattacks worldwide, as attackers exploit weaknesses in software to gain unauthorized access, steal data, or disrupt services. In this study, we evaluated existing research approaches and propose SemVul, a vulnerability detection pipeline that demonstrates better generalization and higher accuracy in learning vulnerable code patterns. We propose a Code Property Graph-based vulnerability-detection approach combined with semantic-level enhancement, enabling the model to capture both the program's structural flow and the semantic meaning of the code. Our approach integrates both node-level and edge-level semantic embeddings using pre-trained code embedding techniques. We systematically evaluate multiple GNN architectures on publicly available benchmark datasets. SemVul is generic with respect to the programming language and supports multiple architectures. By integrating structural and semantic information, the proposed approach improves vulnerability detection performance. Our results show that SemVul outperforms existing approaches and provides better generalization.

1. Introduction

Software vulnerabilities remain a critical threat to modern computing systems. In 2024 alone, over 25,000 new vulnerabilities were reported in the CVE database (Lekssays et al., 2025), while the open-source ecosystem continues its explosive growth with more than 60 million new repositories created on GitHub in 2020 (Song et al., 2022). This scale amplifies the attack surface and enables supply-chain attacks, as exemplified by high-profile incidents like SolarWinds. Vulnerabilities introduced at the source code level propagate across dependent projects, leading to severe security breaches. Preventing such exploits requires accurate and scalable detection early in the development lifecycle.

Existing approaches, such as static analysis, dynamic analysis, and symbolic execution, provide formal guarantees for certain vulnerability classes but suffer from well-known limitations. Static analyzers produce high false-positive rates and rely on expert-written rules, while dynamic approaches impose significant computational overhead, unsuitable for continuous integration pipelines (Marjanov et al., 2022). This has motivated growing interest in data-driven techniques that learn vulnerability patterns directly from source code.

1.1. Achievements and limitations of prior work

We categorize existing learning-based approaches into three paradigms, each with distinct strengths and weaknesses.

Token-based sequence models are used by methods such as VulDeePecker (Z. Li et al., 2018), SySeVR (Z. Li et al., 2021), and LineVul (Fu and Tantithamthavorn, 2022), which treat code as token sequences processed by neural networks. LineVul leverages CodeBERT pretraining (Feng et al., 2020) to achieve strong results on curated benchmarks, capturing contextual token patterns that correlate with vulnerabilities. However, these models fundamentally ignore program structure. Recent evaluations reveal that accuracy drops by more than 50% when synthetic and duplicated samples are removed (Chakraborty et al., 2021). The Real-Vul benchmark shows precision losses up to 95% when evaluating on full projects rather than isolated functions (Chakraborty et al., 2024). This suggests token-based models learn superficial lexical patterns rather than meaningful vulnerability semantics.

A second approach is based on **graph-based structural models**. Devign (Zhou et al., 2019) pioneered using Code Property Graphs (CPGs) with gated graph neural networks, demonstrating significant

[☆] Editor: Hongyu Zhang.

* Corresponding author.

E-mail addresses: younas.affan@unica.it (A. Younas), leonardo.regano@unica.it (L. Regano), giacinto@unica.it (G. Giacinto).

gains over sequence models. Follow-up work extends this: HGVul (Song et al., 2022) uses heterogeneous graphs with attention mechanisms; DeepWukong (Cheng et al., 2021) applies program slicing; IVDetect (Y. Li et al., 2021) combines control-flow, data-flow, and call graphs. These methods consistently outperform token-based baselines by capturing structural program dependencies. However, graph-based models often underutilize semantic code content. By relying on one-hot node type encodings or simple feature engineering, they miss the rich contextual meaning encoded in identifiers, literals, and API names. Additionally, GNNs face challenges with over-smoothing (Q. Li et al., 2018) and over-squashing, limiting their ability to propagate vulnerability-relevant signals across complex graphs.

Finally, another approach leverages **Large Language Models**. Indeed, LLMs like GPT-5 and CodeLlama offer impressive code understanding capabilities, potentially leveraging vast pretraining knowledge to identify vulnerability patterns. However, comprehensive evaluations caution against relying too heavily on LLMs for security tasks. The SecLLMHolmes framework (Ullah et al., 2024) found non-deterministic responses and high false-positive rates across eight LLMs. Simple changes such as renaming variables caused PaLM2 and GPT-4 to misclassify 26% and 17% of examples, respectively. Furthermore, the PrimeVul benchmark (Ding et al., 2024) demonstrated that GPT-4's vulnerability predictions perform *worse than random guessing* on paired evaluation, with balanced accuracy of only 46.3% below the 50% random baseline. These results reveal fundamental brittleness that makes LLMs unreliable for security-critical applications.

1.2. Our approach: Semantic-enhanced graph learning

Motivated by these observations, we propose **SemVul**, a semantic-enhanced vulnerability detection pipeline that addresses the limitations of both token-based and purely structural approaches. Our key insight is that effective vulnerability detection requires *both* structural understanding of program flow *and* semantic understanding of code content.

As illustrated in Fig. 1, SemVul operates as follows:

- 1. Code Property Graph Extraction:** We use Joern (Yamaguchi et al., 2014) to parse source functions into CPGs that unify AST, CFG, and DFG representations into a single heterogeneous graph. Each CPG node corresponds to a source code element (statement, expression, or declaration), with source code text extracted from the CPG's CODE property.
- 2. Semantic Embedding Integration:** Each CPG node contains source code text (extracted from the graph's CODE property), which we transform into dense vector representations using pre-trained code models (CodeBERT, GraphCodeBERT, UniXcoder) or dataset-specific Word2Vec. This captures the semantic meaning of identifiers, API calls, and literals information invisible to purely structural approaches. The CPG edges then serve to *correlate* these source code embeddings: data-flow edges connect statements where a variable's definition reaches its use, control-flow edges link statements in execution order, and syntactic edges preserve hierarchical code structure.
- 3. Edge-Type-Aware GNN Processing:** We employ GNN architectures that explicitly model different edge types (syntax, control-flow, data-flow), allowing the model to learn which relationships between source code lines are most relevant for detecting specific vulnerability patterns. For instance, data-flow edges help trace tainted values from input to dangerous operations, while control-flow edges reveal missing safety checks.
- 4. Multi-Statistic Graph Pooling:** We aggregate node representations into a graph-level vector using pooling strategies that capture distributional properties (mean, max, standard deviation), providing richer signals than single-statistic approaches.

Throughout the rest of the paper we use the function of Fig. 2 as a running example. The function copies an attacker-controlled parameter input into a fixed ten-byte buffer via `strcpy`, a classic buffer overflow (CWE-120). Joern turns it into a CPG with seven nodes (METHOD, PARAM input, LOCAL buffer, CALL `strcpy`, its two ARG nodes, and RETURN) connected by three kinds of typed edges: AST edges give the syntactic backbone, CFG edges order the declaration, the call, and the return, and DFG edges carry the vulnerability-relevant fact that input flows unmodified into the second argument of `strcpy`.

Our approach addresses the core limitations identified in prior work: by combining CPG structure with code embeddings, we capture both “how data flows” (structure) and “what the code means” (semantics). A dangerous function call like `strcpy` is recognized both by its position in the data-flow graph and by its semantic embedding.

Furthermore, unlike token-based or LLM approaches that are sensitive to variable renaming, our structural representation remains stable under cosmetic code changes. The CPG topology is preserved even when identifiers are renamed.

Finally, different vulnerability types manifest in different program aspects. Buffer overflows are visible in data flow; null pointer dereferences appear in control flow. We employ multi-relational GNN architectures (RGAT, HGT, MultiView), able to learn which edge types are most discriminative for a given vulnerability pattern.

1.3. Contributions

This paper makes the following contributions:

- 1. Semantic-Enhanced CPG Learning:** We propose integrating pre-trained code embeddings with Code Property Graphs, enabling GNNs to jointly model structural program flow and semantic code meaning.
- 2. Comprehensive Architecture Comparison:** We systematically evaluate multiple GNN architectures (GGNN, RGAT, HGT, MultiView, Call-Aware) and embedding strategies (Word2Vec, CodeBERT, UniXcoder) to understand their contributions to detection performance.
- 3. Reproducible End-to-End Pipeline:** We provide a complete, reproducible pipeline from raw source code to trained models, with explicit handling of data quality issues including deduplication and cross-split leakage prevention.
- 4. State-of-the-Art Results:** SemVul achieves competitive or superior performance on multiple benchmarks, outperforming prior methods.
- 5. Language-Agnostic Architecture:** While our experiments focus on C/C++ code, the pipeline architecture is language-agnostic. Joern supports parsing of multiple programming languages (Java, JavaScript, Python, Go, among others), and our graph construction and GNN components require no language-specific modifications. Extending to other languages requires only re-training embeddings and regenerating CPGs the model architecture remains unchanged.

1.4. Research questions

Our study addresses the following research questions:

- RQ1 (Semantic Enhancement):** *To what extent do pre-trained code embeddings (CodeBERT, GraphCodeBERT, UniXcoder) improve vulnerability detection effectiveness measured by F1 score (balance between correctly identifying vulnerabilities and avoiding false alarms) and AUC (ability to rank vulnerable code higher than safe code) over baseline representations (one-hot node types, dataset-specific Word2Vec)?*

We hypothesize that pre-trained embeddings capture richer semantic information than dataset-specific or structural-only representations. We measure the absolute improvement in F1 and

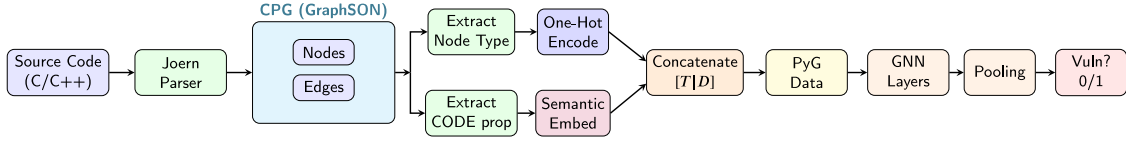


Fig. 1. SemVul system architecture. Source code is parsed by Joern into a Code Property Graph (CPG) stored as GraphSON. Each CPG node contains properties including CODE (source code text) and label (node type). We extract node types for one-hot encoding (T dimensions) and CODE properties for semantic embedding (D dimensions). These are concatenated to form node features, combined with graph structure into PyG Data, processed by GNN layers, pooled to graph-level, and classified as vulnerable or safe.

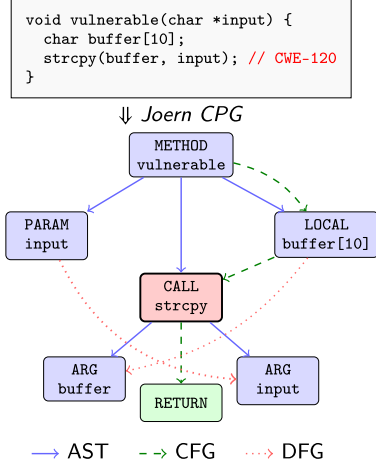


Fig. 2. Code-to-graph transformation for buffer overflow (CWE-120). Untrusted input flows via DFG edges to `strcpy`, which writes to a fixed buffer.

AUC when replacing one-hot encodings with progressively richer embeddings.

- **RQ2 (GNN Architecture):** Among GGNN, RGAT, HGT, MultiView, and Call-Aware architectures, which achieves the best vulnerability detection performance (highest F1 score) on each benchmark, and do edge-type-aware models consistently outperform edge-agnostic baselines?

We systematically compare architectures across four datasets (Devign Zhou et al., 2019, Reveal Chakraborty et al., 2021, Draper Russell et al., 2018, PrimeVul Ding et al., 2024) to determine whether explicit edge-type modeling provides consistent benefits. We also examine whether architecture choice interacts with dataset characteristics (e.g., class imbalance, graph size).

- **RQ3 (Cross-Dataset Transfer):** What is the degradation in vulnerability detection capability (measured by F1 drop) when models trained on one dataset are evaluated on another, and which architectural choices minimize this degradation?

We evaluate cross-dataset transfer by training on each dataset and testing on the others. This quantifies the generalization gap caused by schema heterogeneity (15–16 node types, 18–19 edge types) and distribution shift, providing practical guidance for deployment scenarios.

- **RQ4 (Multi-Dataset Generalization):** Does SemVul maintain consistent vulnerability detection performance across datasets with diverse characteristics (balanced vs. imbalanced, noisy vs. clean labels, varying graph sizes), unlike prior approaches that excel on one benchmark but degrade on others?

Many existing vulnerability detection methods report strong results on a single dataset but fail to generalize. We evaluate SemVul on four benchmarks with distinct properties Devign (balanced, 45% vulnerable), Draper (imbalanced, 6%), Reveal (imbalanced, 9%), and PrimeVul (highly imbalanced, 3.5%, clean labels) to demonstrate consistent effectiveness across diverse evaluation settings.

The remainder of this paper is organized as follows: Section 2 reviews background and related work; Section 3 presents our methodology; Section 4 describes the experimental setup; Section 5 presents results; and Section 6 concludes with future directions.

2. Background and related work

Software vulnerability detection has evolved from manual audit and rule-based static analysis toward data-driven approaches that can scale to large codebases. Static analyzers relied on hand-crafted patterns over abstract syntax tree (AST) or control-flow graph (CFG) fragments, which delivered high precision for specific weaknesses but struggled with recall and generalization (Marjanov et al., 2022). The rise of machine learning introduced representations that encode syntactic, semantic, and control-flow context as features for classifiers. In vulnerability detection, this shift is driven by two needs: capturing long-range dependencies in code and modeling program semantics that are often invisible in surface-level token patterns (Marjanov et al., 2022; Chakraborty et al., 2021). Our work follows this trajectory by using Code Property Graphs (CPGs) and multi-source embeddings to model both structure and content.

2.1. Program representations for vulnerability detection

Program representation is central to vulnerability detection. Representations include abstract syntax trees (ASTs), control-flow graphs (CFGs), program dependence graphs (PDGs), and data-flow graphs. Each captures a different aspect of program behavior: ASTs provide syntax structure, CFGs capture execution paths, and data-flow graphs highlight variable dependencies.

Code Property Graphs unify these views into a single, typed, multi-edge graph by merging syntax, control, and data dependencies (Yamaguchi et al., 2014). CPGs are a natural fit for graph learning because they expose a rich set of nodes, edges, and attributes that can be mapped to discrete vocabularies or vector embeddings. Tools such as Joern automate the extraction of CPGs from source code and have become the de facto standard in security research. In our pipeline, Joern is used to parse per-function code and export GraphSON graphs, enabling consistent downstream processing across datasets. CPGs encode different edge classes syntax edges, control-flow edges, data-dependence edges, and call relationships which can be assigned typed embeddings or one-hot encodings. Several studies show that explicitly modeling edge types improves vulnerability detection because control and data dependencies convey the flow of potentially unsafe operations (Song et al., 2022; Nguyen et al., 2025).

A key foundation for this line of work is Devign (Zhou et al., 2019), which introduced a graph-based approach to data-driven vulnerability detection at the function level. Devign models source code functions as composite graphs that integrate multiple program representations abstract syntax trees, control-flow graphs, data-flow graphs, and natural code sequences into a unified structure with heterogeneous edge types. The approach employs gated graph neural networks (GGNNs) to learn node-level representations through message passing, followed by a convolutional module to aggregate graph-level features for binary classification of vulnerability status. Devign demonstrated that

combining structural program analysis with deep learning significantly improves detection accuracy over purely sequential or token-based methods. This dataset has shaped subsequent research because it offers a consistent experimental setting and because its function-level focus aligns with practical code review tasks. In our study, Devign serves both as a baseline and as a reference point for the improvements introduced by CPG-driven representations and embedding fusion.

Another relevant thread in the literature compares static and dynamic analysis for vulnerability detection. Dynamic approaches can capture runtime behavior but require test inputs and impose significant overhead, which limits scalability across large repositories, whereas static approaches must approximate program behavior and can miss path-sensitive issues (Marjanov et al., 2022). CPG-based analysis provides a middle ground by encoding control and data dependencies in a static graph, allowing learning methods to reason about potential flows without executing code.

2.2. Graph neural networks for code

Graph-based vulnerability detection builds on message passing over program graphs. Graph neural networks learn representations by iteratively aggregating information from neighbors, allowing them to encode localized and long-range dependencies. Prior work explores multiple architectures including graph convolutional networks (GCNs) (Kipf and Welling, 2016), graph attention networks (GATs) (Veličković et al., 2017), GraphSAGE (Hamilton et al., 2017), graph isomorphism networks (GINs) (Xu et al., 2018), gated graph neural networks (GGNNs) (Li et al., 2015), and heterogeneous or hypergraph models that handle typed edges (Marjanov et al., 2022; Song et al., 2022). The Devign model itself uses a gated graph neural network to classify functions, while later work introduces hierarchical pooling methods such as DiffPool (Ying et al., 2018) and SAGPool (Lee et al., 2019), edge-type attention and semantic enhancements (Zhou et al., 2019; Nguyen et al., 2025).

In our architecture, the graph model is configurable with multiple GNN layers, edge-aware operators, and pooling variants such as multi-statistic pooling to capture global structure. These design choices align with the literature that emphasizes the importance of edge types, node types, and hierarchical composition in vulnerability prediction (Nguyen et al., 2025; Song et al., 2022).

2.3. Pre-trained code models

Parallel to graph learning, transformer-based models have reshaped code understanding. Models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), CodeT5+ (Wang et al., 2023), and UniXcoder (Guo et al., 2022) are pretrained on large corpora with objectives tailored to code, achieving strong results in tasks like function name prediction, code search, and defect detection. These models capture token-level semantics and long-range dependencies that are difficult to express in graph-only approaches (Marjanov et al., 2022). However, they typically operate on linearized code sequences and can underutilize explicit program structure. In vulnerability detection, transformer embeddings are often used as inputs to classifiers or as features attached to graph nodes.

The vocabulary and embedding pipeline also builds on related work in representation learning. Existing code embeddings use word2vec (Mikolov et al., 2013) or Code2Vec (Alon et al., 2019) to capture token co-occurrence and path-based code representations, while newer approaches use subword tokenization and transformer encoders. For graph-based models, vocabularies are built for node types, edge types and content strings, and embeddings are cached to scale to large datasets.

Over the last three to four years, research on automated vulnerability detection has converged on graph representations of code as its structural backbone, with the Code Property Graph (CPG) extracted by

the Joern platform appearing across nearly every branch of the literature. A recent large-scale systematic review of 138 studies published between 2011 and mid-2024 confirms this picture quantitatively: graph-based input representations dominate the field at 57.2% of the surveyed studies, deep learning models account for 88.4% of them with graph neural networks among the most popular architectures, and Joern is the single most widely used code representation tool (Shiri Harzevili et al., 2024). Building directly on this foundation, one study converts source code into CPGs, vectorizes the nodes with CodeBERT, enriches them with graph centrality features, and classifies with an adaptive graph neural network that combines Transformer-style attention with graph convolutions (Liang et al., 2024). A complementary direction targets the graph structure itself rather than the node features: AMPLE (Wen et al., 2023) first simplifies the code structure graph, merging syntax nodes of the same type and linking nodes that refer to the same variable so as to reduce graph size and shorten the distance between related statements, and then learns over the simplified graph with an edge-aware graph convolutional network and a kernel-scaled representation module that balances local and distant dependencies. Going a step further, another line of research argues that language models and GNNs cover each other's blind spots and fuses them explicitly: pre-trained code language model embeddings initialize the nodes of gated GNNs over CPGs, an online knowledge distillation scheme propagates structural information across layers, and a late linear interpolation combines the two models; evaluated on four real-world datasets against seventeen state-of-the-art approaches, this fusion yields roughly 10% F1 gains on challenging imbalanced benchmarks (Liu et al., 2025). The most recent studies keep the graph but move the reasoning into large language models. One such study shows that LLMs applied to plain source text overlook the syntactic and semantic structure of code, and therefore injects CPG-derived structural information into the code representation while retrieving in-context demonstrations by jointly weighing semantic, lexical, and syntactic similarity; on the Devign, Reveal, and Big-Vul datasets this approach outperforms six state-of-the-art baselines (Lu et al., 2024). In a similar spirit but at larger scale, a follow-up study observes that state-of-the-art detectors lose up to 45% accuracy on rigorously verified datasets and degrade under simple code edits, and responds by using the CPG as a guide rather than an input: Joern queries extract potential vulnerable execution paths and backward slicing condenses them into focused snippets, shrinking the code shown to the LLM by 67.84% to 90.93% while preserving vulnerability-relevant context, which enables project-level analysis and delivers 15%–40% F1 improvements on verified datasets together with robustness to syntactic modifications (Lekssays et al., 2025). Taken together, these latest studies show that whether the downstream model is a GNN, a fused architecture, or an LLM, a Joern-derived graph is still computed for every function or slice, so the construction and cost of CPG-based pipelines remains shared, an important part for the entire research direction.

While these recent works demonstrate the value of CPGs for vulnerability detection, they either use the graph structure in isolation, pair it with a single fixed semantic component, or delegate the reasoning entirely to a large language model. Our work differs in both mechanism and scope. At the mechanism level, SemVul enriches every CPG node with a precomputed semantic embedding of its code content, fused with the structural encoding before message passing, so that semantics and structure are reasoned over jointly rather than combined at the decision level; because the embeddings are precomputed, this comes without the cost of jointly training a language model. At the scope level, and to the best of our knowledge, SemVul is the first to systematically quantify the contribution of node-level semantic enrichment across multiple embedding models (from Word2Vec to CodeT5+), multiple relation-aware GNN architectures, and four benchmarks, including the rigorously labeled PrimeVul, rather than proposing a single fixed combination. This controlled design lets us attribute performance to individual components, a question the works above leave open.

2.4. Hybrid approaches

Hybrid approaches that fuse graphs and language models are increasingly common. Some methods use transformers to encode code tokens and then map them onto graph nodes, while others perform late fusion by combining graph-level embeddings with sequence-level representations. The choice of fusion strategy affects training stability and interpretability. In our pipeline, embeddings are extracted for both node content and edge content and merged with structural features through early concatenation at the node level, which, as we will argue in Section 3.3, is the fusion strategy dictated by the interdependence of semantic and structural information in this setting. This design is motivated by prior work showing that textual content alone can miss control or data dependencies, while pure structural models can miss semantic cues (Nguyen et al., 2025). By integrating both, we target a representation that is robust to syntactic variation and retains contextual semantics critical for identifying vulnerabilities.

Recent work also explores self-supervised and contrastive objectives for code representations. Contrastive learning (Khosla et al., 2020) can align augmented views of the same function, or align graph and token representations, improving downstream classification when labeled data is limited. Approaches such as graph contrastive learning and masked token prediction aim to capture invariances that reflect program semantics rather than surface syntax. These techniques are particularly relevant in vulnerability detection, where labels are noisy and vulnerability patterns can be subtle (Ji et al., 2024).

2.5. Dataset quality and evaluation challenges

Dataset preparation is a critical component in vulnerability detection research. Vulnerability datasets often suffer from duplicates, near-duplicates, or split contamination that inflates validation performance. Hash-based deduplication is a common remedy, but the hashing strategy must reflect the actual code content to avoid mismatches (Chakraborty et al., 2021, 2024).

Another issue is class imbalance. Real-world code has far more non-vulnerable functions than vulnerable ones, so metrics like accuracy can be misleading. Prior work emphasizes the use of area under the ROC curve (AUC) (Bradley, 1997), precision, recall and F1-score (Sokolova and Lapalme, 2009) to capture detection quality (Marjanov et al., 2022). Many approaches also introduce class-balanced loss functions, focal loss variants or label smoothing to improve training stability.

An important challenge is cross-dataset generalization. Devign and Reveal are curated datasets with different labeling protocols and function extraction processes, while Draper includes broader CWE annotations and distinct code sources. Models that perform well on one dataset often degrade on another, highlighting the influence of dataset composition and preprocessing (Chakraborty et al., 2024). Our pipeline is designed to minimize these confounds by standardizing preprocessing, per-function splitting, and graph extraction across datasets. This standardization enables more meaningful cross-dataset comparisons and supports claims about generalization and robustness.

Finally, another issue relates to the co-existence of different graph export formats, which expose slightly different fields. Mismatches between the graph schema and model assumptions can silently degrade performance. GraphSON, for example, contains typed nodes and edges with property dictionaries, but the mapping from properties to model features requires careful normalization. Our pipeline explicitly builds vocabularies for node types, edge types, and content strings, and maintains configuration files that mirror the extracted schema.

3. Methodology

This section describes our approach to semantic-enhanced vulnerability detection. We first formalize the problem, then present our code representation strategy that combines structural graphs with semantic embeddings, and finally describe our GNN architectures. Data preprocessing and implementation details are deferred to Section 4.

Fig. 1 summarizes our complete pipeline: source code is parsed into CPGs, enriched with semantic embeddings, processed by GNNs that exploit edge-type heterogeneity, pooled to graph-level representations, and classified. This design addresses the limitations of prior work by combining structural program understanding with semantic code meaning. Figs. 3 and 4 provide a more detailed depiction of our pipeline, showing respectively the initial feature construction executed per each CPG node, and the subsequent data flow leading to the predictions.

Returning to the running example, Fig. 3 shows how the CALL strcpsy node of Fig. 2 becomes a feature vector. Path A extracts its label CALL and one-hot encodes it into T dimensions ($T = 15$ for Devign), a vector that would be identical for a call to strncpsy. Path B extracts its code content strcpsy, normalizes and hashes it, and retrieves its pre-computed D -dimensional semantic embedding from the Embedder (e.g., $D = 768$ for GraphCodeBERT); this is the component that separates strcpsy from strncpsy. The two blocks are concatenated into the final $T+D$ node feature, and the same procedure is applied to the remaining six nodes, yielding the matrix `data.x` of shape $[7, T+D]$.

Fig. 4 completes the running example by following the graph of Fig. 3 to a prediction. In Stage 1 the function is a PyG object with `data.x` of shape $[7, T+D]$, an `edge_index` listing the source and destination of every AST, CFG, and DFG edge in Fig. 2, and an `edge_type` vector recording each edge's relation. Stage 2 splits each node feature back into its one-hot and semantic blocks and maps them to a unified H -dimensional hidden state. In Stage 3, edge-type-aware message passing propagates information along the typed edges: through the relation-specific weight of the DFG relation, the CALL strcpsy node receives the representation of PARAM input via ARG input, while no bounds-checking node exists on the control-flow path to attenuate it; after L layers the call node therefore encodes the joint pattern "risky API receiving unchecked, attacker-reachable data". Stage 4 pools the node embeddings into one graph vector and the MLP classifier maps it to the positive (vulnerable) class.

3.1. Problem formulation

We formulate vulnerability detection as a binary graph classification problem. Given a function f represented as a Code Property Graph $G = (V, E, X, \mathcal{T}_v, \mathcal{T}_e)$, where $V = \{v_1, \dots, v_n\}$ is the set of nodes representing code elements, $E \subseteq V \times V$ is the set of directed edges representing relationships, $X \in \mathbb{R}^{n \times d}$ is the node feature matrix, $\mathcal{T}_v : V \rightarrow \{1, \dots, K\}$ maps nodes to K node types, and $\mathcal{T}_e : E \rightarrow \{1, \dots, R\}$ maps edges to R relation types. The goal is to learn a function $\Phi : G \rightarrow \{0, 1\}$ that predicts whether the function contains a vulnerability.

3.2. Code property graph representation

We use Joern (Yamaguchi et al., 2014) to generate Code Property Graphs that unify multiple program representations into a single heterogeneous graph:

$$G_{\text{CPG}} = G_{\text{AST}} \cup G_{\text{CFG}} \cup G_{\text{DFG}} \cup G_{\text{PDG}} \quad (1)$$

where G_{AST} captures syntactic structure, G_{CFG} captures execution order, G_{DFG} captures data dependencies, and G_{PDG} captures program dependencies.

CPG nodes represent different code elements, each with distinct roles in vulnerability patterns. The source code text for each node is extracted from the CPG's CODE property, which we later transform into

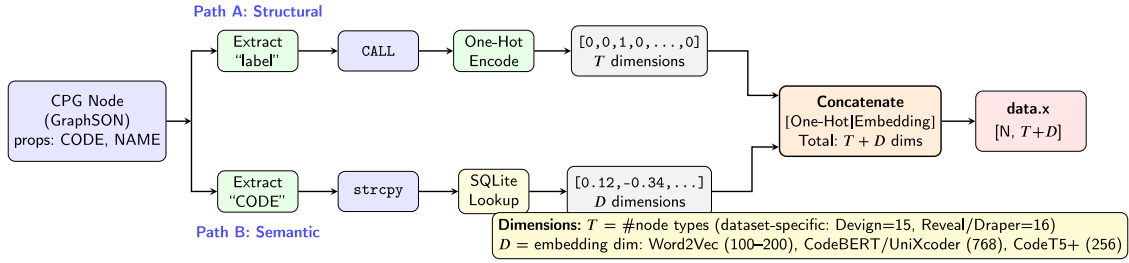


Fig. 3. Feature construction pipeline for each CPG node. **Path A (Structural):** Node type (e.g., CALL, IDENTIFIER) is converted to a sparse one-hot encoding (T dimensions, where T is dataset-specific: Devign=15, Reveal=16, Draper=16). **Path B (Semantic):** Node content (source code text) is normalized, hashed with SHA256 for efficient lookup, and the pre-computed embedding (D dimensions) is retrieved from SQLite cache. Both vectors are concatenated to form the final node feature vector ($T+D$ dimensions). Embedding dimension D varies by model: Word2Vec (100–200), CodeBERT/UniXcoder (768), CodeT5+ (256).

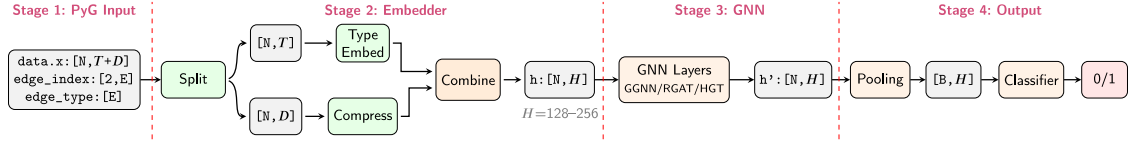


Fig. 4. Complete data flow from input to prediction. **Stage 1: PyG Data** contains node features `data.x` of shape $[N, T+D]$ (one-hot type encoding concatenated with semantic embedding), graph structure `data.edge_index` of shape $[2, E]$ where the two rows store source and destination node indices respectively for each of the E edges (COO sparse format), and relation types `data.edge_type` of shape $[E]$ storing the edge type index for each edge. **Stage 2: The embedder** splits the concatenated input back into its one-hot block (T dims) and semantic block (D dims). The one-hot block is converted to learned dense embeddings, while the semantic block is compressed to a lower dimension. Both outputs are concatenated and passed through a linear layer to produce unified hidden representations $[N, H]$. **Stage 3: GNN** performs L rounds of edge-type-aware message passing using relation-specific weights W_r , producing updated node embeddings h' . **Stage 4: Pooling** (e.g., mean, max, attention, or multi-stat) aggregates per-node embeddings to a single graph-level vector $[B, H]$, and the classifier (a two-layer MLP with LayerNorm, GELU, and dropout) maps this to a binary vulnerability prediction. Symbols: N = nodes per graph, E = edges per graph, B = graphs per batch, T = node-type dimensions, D = embedding dimensions, H = hidden dimensions.

semantic embeddings (see Section 3.3). Key node categories include **Function Call Nodes** (invocations of functions, the most vulnerability-critical node type since dangerous functions like `strcpy`, `sprintf`, and `gets` are represented as call nodes), **Identifier Nodes** (variable references that track data flow from tainted sources), **Literal Nodes** (constant values including buffer sizes, e.g., `char buf[10]`), **Control Structure Nodes** (conditionals and loops whose presence (or absence) determines whether safety checks exist), **Declaration Nodes** (local variable declarations including buffer allocations), and **Entry/Exit Nodes** (function entry and return points, relevant for resource leak detection and scope analysis). The node type vocabulary varies across datasets: Devign has 15 node types while Reveal and Draper have 16 (including struct member access patterns). This schema heterogeneity means models must adapt their type encoding dimensions per dataset. The complete schema for each dataset is provided in Appendix.

The CPG edge types carry distinct semantic meanings critical for vulnerability detection. Each edge category corresponds to one of the underlying program representations (AST, CFG, DFG, PDG) introduced in Section 2. **Data-Flow Edges** (from G_{DFG}) show where a variable's value reaches its use, critical for tracking tainted data from sources (e.g., user input) to sinks (e.g., unsafe copy functions). **Control-Flow Edges** (from G_{CFG}) encode execution order between statements, and are thus essential for detecting missing bounds checks or null guards. **Syntax Edges** (from G_{AST}) represent parent-child relationships in the abstract syntax tree, capturing code structure but less directly vulnerability-relevant. **Control-Dependence Edges** (from G_{PDG}) show which statements are conditionally executed, thus are important for understanding branch coverage of safety checks. **Dominance Edges** represent relationships in control flow identifying mandatory execution paths. Finally, **Call-Related Edges** encode function call and argument relationships, and are critical since dangerous library functions are often sites of vulnerabilities. Different vulnerability types manifest through different edge patterns: buffer overflows appear in data-flow chains to unsafe functions, while null pointer dereferences appear as control-flow paths lacking protective conditionals.

CPGs provide three key advantages over token sequences. First, vulnerabilities often involve data flowing from untrusted sources to sensitive sinks. Data-flow edges explicitly encode these dependencies. Second, missing bounds checks or null pointer guards are visible in control-flow structure the path to a dangerous operation lacks a protective branch. Third, the graph topology remains stable under variable renaming, comment changes, or whitespace modifications that confuse token-based models. Fig. 2 illustrates this transformation for a buffer overflow vulnerability. The CPG captures how untrusted input flows through the program structure to a dangerous function call information that is invisible to token-based models.

3.3. Semantic-enhanced node representation

A key contribution of SemVul is enriching CPG nodes with semantic embeddings. Each node v_i is represented by $\mathbf{x}_i = [\mathbf{t}_i \parallel \mathbf{e}_i]$, where $\mathbf{t}_i \in \{0, 1\}^K$ is a one-hot encoding of the node type (one of K categories such as function call, identifier, or literal), and $\mathbf{e}_i \in \mathbb{R}^{d_e}$ is a semantic embedding of the node's code content.

We explore multiple embedding sources. First, we train Word2Vec models (Mikolov et al., 2013) on the code vocabulary extracted from each dataset. Tokens are split on CamelCase and snake_case boundaries, and embeddings are aggregated via mean pooling. Then, for richer semantic understanding, we use embeddings from CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), UniXcoder (Guo et al., 2022), and CodeT5+ (Wang et al., 2023). These models are pre-trained on large code corpora with general-purpose objectives, i.e., they are not specifically trained for vulnerability detection purposes. We therefore hypothesize that the semantic representations they produce also encode contextual patterns correlated with vulnerability. This is an expectation that must be verified empirically, which is precisely the aim of RQ1.

The pre-trained code models are frozen and are never fine-tuned. Each model is applied in a single forward pass (inference mode) to precompute embeddings for the code content of the CPG nodes. The resulting vectors are cached and later loaded as fixed node features,

so the language model itself is not part of the trained network and is not even loaded during training. The same holds for Word2Vec, with one difference in provenance. Following the original Devign protocol, we train it ourselves on the code tokens of each dataset, but using only the standard unsupervised objective and without any vulnerability labels, after which it is likewise frozen and used only to look up vectors. Because the embeddings enter the model as fixed inputs rather than trainable parameters, they receive no gradient at any point. The reported semantic enhancement therefore reflects the information already present in these representations, and not any task-specific adaptation of the underlying language models.

The semantics has to be seen as a complement to the structure, rather than as an independent source of information: identical structural patterns may correspond to entirely different vulnerability behavior depending on their semantic content, and conversely the semantics of a code fragment is only meaningful in relation to the structure it belongs to. This logical correlation corresponds to the early fusion paradigm, in which different measures are stacked together to create the feature vector. Late fusion, in contrast, is appropriate when separate sources of information can each independently support the decision, which is not the case here: no decision on the vulnerability of the code could be taken from the semantics alone. For this reason, node-level semantic embeddings are concatenated with the structural one-hot type encoding to form a single node feature vector.

Following, we use an example to explain intuitively why semantic embeddings complement structure. Consider a node representing the call `strcpy(buffer, input)`. With one-hot encoding alone, this node is indistinguishable from any other CALL node, `strcpy` and `strncpy` receive identical feature vectors, so no downstream model can treat them differently. Semantic embeddings remove this bottleneck in three steps. First, pre-trained models such as CodeBERT make the two calls *distinguishable* trained with masked-token objectives on large corpora, they assign different vectors to tokens used in different contexts, and `strncpy` systematically co-occurs with length arguments and `sizeof` expressions while `strcpy` does not. We stress that the embedding encodes no notion of safety; it only guarantees that the classifier receives different inputs for the two calls. Second, the *association with vulnerability is learned during supervised training*: since vulnerable functions contain unchecked `strcpy`-like calls far more often, the GNN learns to map that region of the embedding space to higher risk. Third, the *CPG structure determines whether the risk is realized*: data-flow edges reveal whether `input` originates from an unvalidated source, and control-flow edges reveal whether a bounds check precedes the call, so message passing combines the call's semantics with its context and the model learns the joint pattern "risky API with unchecked input" rather than the mere presence of a token. This also explains our early-fusion choice. The decision emerges from semantics and structure jointly, and neither is alone sufficient.

3.4. Graph neural network architectures

All our GNN architectures follow the Message Passing Neural Network framework (Gilmer et al., 2017), where each node iteratively gathers messages from its neighbors, aggregates them using a permutation-invariant function (sum, mean, or max), and updates its representation. After L layers, each node encodes information from its L -hop neighborhood. We evaluate four architectures. The selection of the four architectures was driven by the nature of our input data rather than by a desire for architectural variety. A Code Property Graph is not a plain graph: it is a multi-relational structure in which nodes carry code semantics and edges belong to distinct types (i.e., AST, control-flow, data-flow), and each edge type carries different information and, mathematically, deserves a different transformation or weight during message passing. This immediately rules out architectures that treat all edges as equivalent, and it defines the design space we sampled from: models that differ in how they exploit typed-edge structure. Within

that space, each of the four architectures represents one distinct and well-motivated hypothesis about how edge-type information should be handled. GGNN assigns each edge type its own weight matrix and uses a gated (GRU) update, testing whether type-specific transformations plus long-range memory are sufficient. RGAT adds learned attention per edge type, testing whether the model benefits from deciding that some relations (for example data-flow when tracking a tainted value) matter more than others for a given node. Multi-View GGNN takes the opposite decomposition: instead of mixing all edge types in one pass, it processes the AST, CFG, and DFG as separate views with separate encoders and fuses them by attention, testing whether vulnerabilities are better captured when each program aspect is modeled independently. Finally, the Call-Aware GNN encodes a domain-specific prior, namely that vulnerabilities frequently manifest at call sites of dangerous functions, and gives CALL nodes and their data-flow context special treatment. Importantly, all four share the same message-passing framework, the same input representation, and the same pooling and classification head, so the comparison is controlled, the only thing that varies is the inductive bias about edge types. Our goal in evaluating them was therefore not to increase model complexity, but to determine empirically which way of handling multi-relational code graphs handles our data best, and to report that comparison rather than a single-architecture result.

Gated Graph Neural Networks (GGNN) (Li et al., 2015) employ edge-type-specific weight matrices to transform neighbor representations differently based on relationship type (e.g., data-flow vs. control-flow edges). The update mechanism uses a Gated Recurrent Unit (GRU), a recurrent neural network component that selectively remembers or forgets information across layers, which is crucial for capturing long-range dependencies such as when a variable defined at the start of a function is used unsafely much later.

Relational Graph Attention Networks (RGAT) (Veličković et al., 2017; Schlichtkrull et al., 2018) combine relational modeling with attention mechanisms. The key insight is that not all neighbors are equally important, and different edge types carry different significance. Attention coefficients are learned per edge type, allowing the model to focus heavily on data-flow edges for tracking tainted values while attending to control-flow edges for detecting missing safety checks.

Multi-View GGNN (Zhuang et al., 2021) decompose the CPG into separate views (AST, CFG, DFG) and processes each independently using separate GGNN encoders, rather than processing all edge types together. The view-specific representations are then combined via learned attention weights, which is particularly effective when different vulnerability types manifest in different program aspects buffer overflows in data flow, missing checks in control flow.

Call-Aware GNN (Cheng et al., 2021) are based on the observation that function calls are where vulnerabilities often manifest. This architecture applies special attention to CALL nodes and explicitly models their data-flow context by aggregating incoming and outgoing data-flow edges. This design is motivated by the fact that dangerous functions like `strcpy`, `memcpy`, and `sprintf` are "choke points" where improper inputs cause vulnerabilities.

3.5. Graph-level pooling and classification

After message passing, we aggregate node representations into a graph-level embedding for classification. Our multi-statistic pooling (Ying et al., 2018) concatenates mean, max, and standard deviation over all node representations, capturing average node behavior, extreme signals that may indicate anomalies, and variability across the graph respectively. This richer distributional representation outperforms single-statistic approaches (e.g., mean-only or max-only pooling) by 3+ F1 points in our experiments. The pooled representation is then passed through a two-layer MLP (multilayer perceptron) classifier with ReLU (Rectified Linear Unit) activation function and sigmoid output for binary vulnerability prediction. We use weighted binary cross-entropy loss with class weights inversely proportional to class frequencies to handle the severe class imbalance present in vulnerability datasets.

Table 1
Dataset statistics.

Dataset	Funks	Nodes	Edges	NT	ET
Devign	27,318	9.65M	68.2M	15	18
Draper	127,437	24.1M	137.3M	16	19
Reveal	22,383	5.69M	38.0M	16	19
PrimeVul	235,021	42.3M	245.8M	16	19

NT = Node Types, ET = Edge Types.

Table 2
Per-function graph statistics (from cache index).

Dataset	Avg. N	Avg. E	Max N	Max E
Devign	243.0	1799.0	12,294	109,293
Draper	189	1078	958	11,598
Reveal	167.9	1165.5	18,682	124,621
PrimeVul	180.2	1045.8	15,421	98,347

4. Experimental setup

This section describes our experimental configuration, including the datasets used, model hyperparameters, and evaluation metrics. All experiments were conducted on an NVIDIA RTX 5000 Ada Generation GPU (32 GB VRAM) with CUDA 13.0 and driver version 580.95.05. Our implementation uses PyTorch 2.0 with PyTorch Geometric (Fey and Lenssen, 2019), Joern version 4.0.399 for CPG extraction. To improve training efficiency, we pre-computed and cached both one-hot node type encodings and semantic embeddings, enabling fast graph construction and data loading during training. We used mixed precision training (FP16) where supported to reduce memory usage. Training times vary by architecture: using EdgeType-GNN as baseline, RGAT requires approximately 1.5× longer due to attention computation, while HGT requires approximately 3× longer due to heterogeneous transformer layers; all experiments completed within 24 h per dataset. Random seeds were fixed at 42 for reproducibility. Class imbalance was handled using weighted loss functions with $\text{pos_weight} = n_{\text{neg}}/n_{\text{pos}}$.

4.1. Datasets

We evaluate our approach on four widely-used vulnerability detection benchmarks. Each dataset contains C/C++ functions labeled as vulnerable or safe. **Table 1** summarizes the key statistics. **Devign** is the most commonly used benchmark for graph-based vulnerability detection. It contains functions extracted from FFmpeg and Qemu with expert-verified labels. **Draper** provides a larger and more diverse collection with CWE annotations. **Reveal** was specifically designed to address data quality issues in earlier datasets by using careful labeling procedures. **PrimeVul** (Ding et al., 2024) is a recent high-quality benchmark with rigorous labeling that addresses the label noise problems in earlier datasets; it is highly imbalanced (3.5% vulnerable), and thus a challenging evaluation setting.

Table 2 shows per-function graph statistics. The variation in graph sizes across datasets reflects differences in function complexity and coding styles. **Table 3** provides detailed per-split statistics including class distribution, which is critical for understanding dataset imbalance. Looking at the statistics, we observe that **Devign is nearly balanced** (~45% vulnerable), making standard metrics meaningful, while **Draper and Reveal are moderately imbalanced** (~6% and ~9% vulnerable respectively), requiring weighted loss functions and careful metric interpretation, and **PrimeVul is extremely imbalanced** (~3.5% vulnerable) with high-quality labels, representing the most challenging and realistic evaluation setting. Also, we observe that **graph sizes vary significantly**: Devign has functions with up to 12,294 nodes, while Reveal reaches 18,682 nodes.

Table 3
Per-split statistics and class imbalance.

Dataset	Split	Funks	Vuln.	%
Devign	Train	21,854	10,018	45.8
	Val	2732	1187	43.4
	Test	2732	1255	45.9
Draper	Train	101,950	6321	6.2
	Val	12,744	812	6.4
	Test	12,743	785	6.2
Reveal	Train	17,974	1697	9.4
	Val	2200	175	8.0
	Test	2209	196	8.9
PrimeVul	Train	188,017	6580	3.5
	Val	23,502	823	3.5
	Test	23,502	822	3.5

Table 4
CPG schema summary by dataset.

Dataset	NT	ET	Notable additions
Devign	15	18	Base schema
Reveal	16	19	MEMBER, CAPTURE
Draper	16	19	Similar to Reveal
PrimeVul	16	19	Similar to Reveal

NT = Node Types, ET = Edge Types.

4.2. Data preprocessing

Reliable vulnerability detection requires careful data engineering to avoid representation mismatch, split contamination, and dataset artifacts. This section describes our preprocessing pipeline.

All datasets are converted into a unified format: each split (train/validation/test) is stored as a directory of per-function C files, associate with a CSV that records function ID, vulnerability label, and content hash. Joern processes each function to produce CPG exports in GraphSON format. This standardization allows us to use the same training code across datasets and ensures that performance differences reflect dataset content rather than preprocessing variations.

CPG extraction produces *different* node and edge type vocabularies depending on the dataset, code patterns, and Joern version. This heterogeneity is a critical observation: **different datasets cannot be treated identically**, as their underlying graph schemas differ. Our pipeline explicitly discovers and validates these schemas. **Table 4** summarizes the schema dimensions for each dataset. The complete list of node and edge types with their descriptions is provided in **Appendix**. This schema heterogeneity has important implications for model design: one-hot dimensions differ across datasets (Devign uses 15-dimensional node type encodings while Reveal/Draper/PrimeVul use 16-dimensional encodings), edge-type-aware GNNs must adapt their weight matrices accordingly, and cross-dataset transfer requires vocabulary mapping and retraining. Our vocabulary builder explicitly discovers these schemas from GraphSON exports and produces synchronized vocabulary files that ensure consistent index assignments within each dataset, preventing silent degradation from missing edge categories or schema drift.

To enable efficient experimentation with different embedding strategies, we pre-compute embeddings for all unique node content strings, and store them in SQLite databases indexed by content hash, enabling fast lookup during dataset construction. In particular, **Word2Vec** is trained per dataset on the extracted code vocabulary, while **pre-trained Models** (CodeBERT, GraphCodeBERT, UniXcoder, CodeT5+) embeddings are computed once and cached. Cache signatures encode the embedding configuration to prevent mismatches.

The final datasets are constructed by loading GraphSON exports for each function, mapping node and edge types to vocabulary indices, concatenating one-hot type encodings with semantic embeddings, and

Table 5

Model configurations.

Model	GNN	Layers	Pool	Embed.
Deign	GGNN	4	Conv1D	learned
Improved	GGNN	1–2	multi_stat	word2vec
Call-Aware	call_aware	4	multi_stat	word2vec
MLP	none	0	multi_stat	codebert
RGAT	rgat	4	Conv1D	learned
MultiView	ggnn	4	Conv1D	learned
HGT	hgt	3	Conv1D	learned

Hidden dimension = $T+D$ for all models.**Table 6**

Node embedding configurations.

Embedder	Dim	Pre-train	Description
One-Hot Only	16	No	Type encoding only
Word2Vec-100d	116	No	Per-dataset trained
Word2Vec-200d	216	No	Per-dataset trained
CodeT5+	272	Yes	16 + 256 dims
CodeBERT	784	Yes	16 + 768 dims
GraphCodeBERT	784	Yes	16 + 768 dims
UniXcoder	784	Yes	16 + 768 dims

Table 7

Training hyperparameters.

Hyperparameter	Value
Optimizer	AdamW
Learning Rate	0.0005
Weight Decay	0.01
Batch Size	64
Maximum Epochs	100 (all models)
Early Stopping Patience	30
Dropout Rate	0.2
Gradient Clipping	1.0
LR Scheduler	ReduceLROnPlateau
Scheduler Patience	10

constructing sparse adjacency representations with edge type labels. The resulting cached datasets support reproducible training across embedding and architecture configurations.

4.3. Model configurations

Table 5 summarizes the configurations for each model variant. We tested both simple baselines (like MLP without GNN) and sophisticated architectures (like RGAT and HGT) to understand the contribution of different components.

We experimented with multiple code embedding strategies to understand their impact on detection performance. Table 6 shows the embedding configurations used in our experiments.

For all models, we set the hidden dimension to the total input feature size ($T+D$), where T is the node-type one-hot size and D is the embedding dimension. For example:

- Deign + Word2Vec-200d uses $T+D = 15+200 = 215$;
- Reveal/Draper/PrimeVul + Word2Vec-200d uses $16+200 = 216$;
- CodeT5+ uses $16+256 = 272$;
- CodeBERT/GraphCodeBERT/UniXcoder use $16+768 = 784$.

All models were trained using the same optimization setup to ensure fair comparison. Table 7 lists the training hyperparameters.

4.4. Evaluation metrics

We evaluate vulnerability detection effectiveness using five standard classification metrics (Sokolova and Lapalme, 2009; Bradley, 1997). All metrics are derived from the four entries of the confusion

matrix computed on the test set: true positives TP (vulnerable functions correctly identified), true negatives TN (safe functions correctly identified), false positives FP (safe functions flagged as vulnerable), and false negatives FN (vulnerable functions missed).

Accuracy measures the overall fraction of correctly classified functions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

Precision is the fraction of functions flagged as vulnerable that are truly vulnerable, and Recall is the fraction of vulnerable functions that are correctly identified:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}.$$

F1 score is the harmonic mean of precision and recall, balancing false positives against false negatives:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}.$$

AUC is the area under the ROC curve, which plots the true positive rate $TPR = TP/(TP + FN)$ against the false positive rate $FPR = FP/(FP + TN)$ as the classification threshold varies:

$$\text{AUC} = \int_0^1 \text{TPR}(\text{FPR}^{-1}(x)) dx.$$

For all metrics, higher values indicate better detection capability. Because vulnerability datasets are typically imbalanced (see Table 3), accuracy alone is misleading: a trivial classifier that predicts “non-vulnerable” for every input achieves high accuracy while detecting no vulnerabilities at all. We therefore rely primarily on F1 and AUC, which remain informative under class imbalance, and report accuracy, precision, and recall to fully characterize each model’s behavior, including the trade-off between false alarms and missed vulnerabilities.

We select the metrics reported for each research question according to its specific goal:

- **RQ1 (Semantic Enhancement)** concerns whether richer representations improve detection. We report Accuracy, F1, and AUC, and center the analysis on the F1 improvement ($\Delta F1$) obtained when replacing structural-only representations with progressively richer semantic embeddings: F1 captures the net effect on both false positives and false negatives, while AUC confirms that the improvement holds across all decision thresholds rather than only at the default operating point.
- **RQ2 (GNN Architecture)** compares architectures to identify the best performer on each benchmark. We use F1 as the primary selection criterion, since it summarizes the balance between false positives and false negatives in a single number at the chosen decision threshold. We additionally report Accuracy, AUC, and Recall: AUC to assess detection performance across all decision thresholds rather than at a single fixed one, and Recall to reveal how much of a given F1 stems from the fraction of vulnerabilities actually caught. On the highly imbalanced PrimeVul benchmark we additionally report Precision, because under such extreme imbalance precision is both the hardest metric to achieve and the most informative about a model’s false-alarm burden — the practical bottleneck for deployment — and it varies substantially across architectures that otherwise reach comparable F1.
- **RQ3 (Cross-Dataset Transfer)** measures degradation under distribution and schema shift. We quantify it using the F1 drop, and equivalently the F1 retention, between in-domain and transfer settings, because F1 is comparable across datasets with different class distributions and is therefore the most meaningful basis for measuring generalization loss.
- **RQ4 (Multi-Dataset Generalization)** asks whether performance is maintained across benchmarks with diverse characteristics. We assess it through F1 consistency across all four datasets, as F1 remains informative under the widely varying class distributions, label-quality levels, and graph sizes that these benchmarks span.

Table 8

RQ1: Structural-only baseline: Graph topology without semantics.

Dataset	Vuln%	Acc	F1	AUC	vs. Rand.
Devign	45.8	54.21	51.37	53.42	+1.37
Draper	6.2	91.84	28.45	62.18	–
Reveal	9.4	87.12	24.83	58.94	–
PrimeVul	3.5	96.12	12.84	54.21	–

Table 9

RQ1: Progressive semantic enhancement (EdgeType-GNN, Devign).

Representation	Dim	Acc	F1	AUC	Δ F1
<i>Structural-Only (No Semantic Content)</i>					
One-Hot Node T.	16	54.21	51.37	53.42	–
<i>Structural + Dataset-Specific Embeddings</i>					
Word2Vec-100d	116	58.34	62.17	64.89	+10.80
Word2Vec-200d	216	58.92	63.84	65.72	+12.47
<i>Structural + Pre-trained Code Model Embeddings</i>					
CodeBERT	784	59.18	65.21	67.45	+13.84
CodeT5+	272	54.62	60.47	62.83	+9.10
GraphCodeBERT	784	59.52	66.01	68.34	+14.64
UniXcoder	784	59.41	65.73	67.91	+14.36

Table 10

RQ1: Semantic enhancement gain across all datasets.

Dataset	Struct-Only		+GraphCodeBERT		Δ F1
	Acc	F1	Acc	F1	
Devign	54.21	51.37	59.52	66.01	+14.64
Draper	91.84	28.45	92.77	53.65	+25.20
Reveal	87.12	24.83	90.04	48.60	+23.77
PrimeVul	96.12	12.84	96.60	23.52	+10.68

5. Results

This section presents our experimental results, organized by research question. The evaluation metrics and their mapping to each research question are defined in Section 4.4. A detailed comparison with prior work is provided in Section 5.6.

5.1. RQ1: Impact of semantic embeddings

Research Question: *To what extent do pre-trained code embeddings improve F1 score and AUC over baseline representations?*

We first establish a critical baseline: **structural-only representation**, which encodes graph topology without semantic code content. This representation used by approaches like Devign (Zhou et al., 2019) consists of one-hot vectors indicating syntactic category (e.g., CALL, IDENTIFIER, LITERAL) from the CPG node label property, and categorical encoding of relationship types (e.g., AST, CFG, REACHING_DEF) from CPG edge labels. However, no semantic content is encoded: the actual code text variable names, function names, literals extracted from the CODE property are discarded.

Table 8 shows the performance of structural-only representation across all four datasets. On the nearly-balanced Devign dataset, structural-only representation achieves **51.37 F1** barely above random chance (50%). This demonstrates that *graph topology alone, without semantic understanding of code content, is insufficient for vulnerability detection*. The model learns which node types connect via which edge types, but cannot distinguish dangerous function calls (e.g., `strcpy`) from safe alternatives (e.g., `strcpy_s`) because both are encoded identically as CALL nodes with identical one-hot vectors. On imbalanced datasets (Draper, Reveal), accuracy appears high (87%–92%) but F1 scores (24%–28%) reveal the model’s failure to identify vulnerable samples it learns to predict “non-vulnerable” for most inputs, exploiting class imbalance rather than learning vulnerability patterns.

Table 9 reports the effect of progressively enriching the node representation on Devign, keeping the architecture fixed to EdgeType-GNN so that the contribution of the embedding is isolated from architectural effects. Each row corresponds to one representation, from the structural-only baseline through dataset-specific Word2Vec to pre-trained code models, and reports its dimensionality together with accuracy, F1, AUC, and the F1 gain over the structural-only baseline (Δ F1). The values show a consistent ordering. Dataset-specific Word2Vec already recovers a substantial part of the gap (+10.80 F1 at 100 dimensions, +12.47 at 200), indicating that even token co-occurrence statistics learned from the dataset itself carry information that one-hot type encoding discards. Pre-trained code models improve further, with GraphCodeBERT achieving the best result (66.01 F1, +14.64), closely followed by UniXcoder (+14.36) and CodeBERT (+13.84); AUC follows the same ordering, confirming that the gain is not an artifact of the decision threshold. The one exception is CodeT5+ (+9.10), which underperforms Word2Vec-200d despite being pre-trained, suggesting that its lower-dimensional (256) representation of short code fragments is less informative in this setting than a larger embedding trained directly on the target vocabulary. Overall, richer semantic representations yield monotonically better detection. This improvement is driven by the semantic information itself rather than by the larger feature size. Table 9 allows the two factors to be separated because performance does not follow dimensionality. CodeT5+ uses 272 dimensions yet reaches only 60.47 F1, which is below Word2Vec-200d at 216 dimensions (63.84) and even below Word2Vec-100d at 116 dimensions (62.17). If additional dimensions were themselves responsible for the gain, a wider representation could not be outperformed by two narrower ones. Moving from the structural-only representation to the smallest semantic one, that is from 16 to 116 dimensions, already yields most of the improvement (+10.80 F1), whereas a further increase of almost seven times in dimensionality adds only around four points. What matters is therefore not the number of added dimensions, but the amount of discriminative information they carry about the code.

Table 10 reports the results of our first research question. It compares, for each of the four datasets, the accuracy and F1-score of the model trained on structural representations alone (Struct-Only, i.e., graph structure and node types without semantic embeddings) against the same model when GraphCodeBERT semantic embeddings are added (+GraphCodeBERT) the last column reports the resulting F1 gain (Δ F1). Two observations emerge from the values. First, the structural-only configuration achieves consistently low F1-scores on all four datasets (51.37 on Devign, 28.45 on Draper, 24.83 on Reveal, and 12.84 on PrimeVul), even where its accuracy appears high (e.g., 96.12 on PrimeVul). This gap between accuracy and F1 arises from class imbalance: a model can reach high accuracy by favoring the majority (non-vulnerable) class while still failing to identify the vulnerable minority, which is exactly what the low F1 reveals. Second, adding semantic embeddings improves the F1-score on every dataset, with gains of +14.64 on Devign, +25.20 on Draper, +23.77 on Reveal, and +10.68 on PrimeVul, alongside consistent (if smaller) accuracy improvements. Since the improvement appears across all four datasets, which differ in size, source, and class balance, we conclude that the limitation of structural-only representations is systematic rather than dataset-specific. Graph structure alone does not carry enough information to separate vulnerable from safe functions, and the semantic content of the code is a necessary complement to it.

Answer to RQ1: Structural-only representations (one-hot node type encoding + edge type indices, without semantic content from the CODE property) perform at **near-random levels** on balanced datasets (51.37 F1 on Devign) and **fail catastrophically** on imbalanced datasets (24–28 F1 on Draper/Reveal). This empirically proves that *graph topology alone cannot distinguish vulnerable from non-vulnerable code* the model cannot differentiate `strcpy` from `strcpy_s` when both are encoded as identical CALL nodes.

Table 11

RQ2: Architecture comparison on Devign.

Architecture	EA	Acc	F1	AUC	Rec
EdgeType-GNN	✓	59.52	66.01	68.34	85.58
Call-Aware	✓	56.63	65.76	65.84	90.68
MultiView GGNN	✓	57.98	64.83	65.32	84.30
GGNN (baseline)	×	56.24	62.45	63.21	81.12

Table 12

RQ2: Architecture comparison on Draper.

Architecture	EA	Acc	F1	AUC	Rec
EdgeType-GNN	✓	92.77	53.65	87.28	64.61
Call-Aware	✓	92.17	52.92	87.47	68.00
RGAT	✓	91.32	49.68	87.36	66.18
MultiView GGNN	✓	91.51	48.67	87.06	62.18
GGNN (baseline)	×	91.32	46.78	85.41	58.91
GAT (baseline)	×	87.40	37.13	83.59	57.45

Table 13

RQ2: Architecture comparison on Reveal.

Architecture	EA	Acc	F1	AUC	Rec
MultiView GGNN	✓	90.04	48.60	84.53	53.06
RGAT	✓	90.40	46.46	82.40	46.94
HGT	✓	89.72	45.30	84.64	47.96
Hypergraph	✓	88.37	45.89	83.74	55.61
EdgeType-GNN	✓	89.45	44.92	82.33	48.47
Call-Aware	✓	88.37	40.09	79.90	43.88

Table 14

RQ2: Architecture comparison on PrimeVul (3.5% vulnerable).

Arch.	Acc	AUC	Prec	Rec	F1
Hypergraph	96.60	81.66	23.78	23.26	23.52
MultiView GGNN	96.59	81.55	21.87	20.15	20.97
RGAT	94.27	81.89	15.25	34.07	21.06
Improved Devign	95.50	79.70	16.81	25.46	20.25
Improved Devign	96.06	81.30	18.34	21.79	19.92
HAGNN	95.14	76.43	16.36	22.53	18.95

Semantic enhancement transforms performance. Using dataset-specific Word2Vec, it provides +10-12 F1 points by capturing token co-occurrence patterns. With pre-trained code models (GraphCodeBERT), it provides +14-25 F1 points by leveraging semantic knowledge from large-scale code corpora, enabling recognition that `strcpy` is semantically related to other dangerous unbounded copy operations.

This confirms our core hypothesis: effective vulnerability detection requires *both* structural understanding (CPG topology showing how code elements relate) *and* semantic understanding (what those code elements mean). Structure without semantics yields near-random predictions.

5.2. RQ2: GNN architecture comparison

Research Question: Among GGNN, RGAT, HGT, MultiView, and Call-Aware architectures, which achieves the highest F1 score, and do edge-type-aware models consistently outperform edge-agnostic baselines?

Tables 11–13 present architecture comparison results on Devign, Draper and Reveal. In each Table, the EA column indicates if the model is edge-type-aware.

Answer to RQ2: Edge-type-aware architectures **consistently outperform edge-agnostic baselines** across datasets. On Draper, EdgeType-GNN achieves 53.65 F1 versus 46.78 for vanilla GGNN (+6.87 F1 points).

However, the **best architecture varies by dataset**: EdgeType-GNN wins on Devign and Draper, while MultiView GGNN wins on Reveal, suggesting that architecture choice should consider dataset characteristics. MultiView’s multi-perspective aggregation may better handle

Table 15

RQ3: Cross-dataset transfer (F1 Score).

Train → Test	Devign	Draper	Reveal	Avg Drop
Devign	66.01	31.24	28.17	−37.13
Draper	42.58	53.65	35.42	−14.87
Reveal	38.92	29.87	48.60	−14.21

Reveal’s higher class imbalance (9.4% vulnerable) and larger graphs (up to 18,682 nodes).

Table 14 reports the architecture comparison on PrimeVul (Ding et al., 2024). We consider PrimeVul the most challenging evaluation setting among our four benchmarks for three reasons. First, it is extremely imbalanced, only 3.5% of its functions are vulnerable, so a model must identify a rare minority class and accuracy becomes uninformative, since a trivial classifier that predicts “non-vulnerable” for every sample already exceeds 96% accuracy. Second, its labels were validated through multiple rounds of expert review and the dataset was deduplicated, eliminating the label noise and data leakage that inflate results on earlier benchmarks. Third, precisely because of this cleaner labeling, prior work has shown that models suffer substantial performance drops when re-evaluated on PrimeVul (Ding et al., 2024), so performance obtained here cannot be attributed to memorizing noisy or duplicated samples.

The values in Table 14 reflect exactly this setting. **Accuracy is misleading**: all models exceed 94% accuracy, but this mirrors the majority-class rate rather than any detection capability, which is why we focus on AUC, precision, recall, and F1. **F1 scores are markedly lower** (18%–24%) than on the other datasets, consistent with PrimeVul’s higher label quality and extreme imbalance rather than with a weakness specific to any single architecture. Among the models, the **hypergraph architecture performs best** (23.52 F1), outperforming fine-tuned code language model baselines (18.05–21.43 F1, as reported in Table 18), which suggests that the higher-order relationships captured by hyperedges help in detecting subtle vulnerability patterns. **RGAT achieves the highest recall** (34.07%), indicating that its per-edge-type attention mechanism recovers more vulnerable samples, although at the cost of precision (15.25%).

Our graph-based models, evaluated under the same rigorous labeling, achieve better performance (up to 23.52 F1 for the hypergraph architecture). The margin is admittedly small, but it is obtained against baselines that were optimized end-to-end for this very benchmark, which suggests that learning from program structure yields representations that generalize beyond memorized lexical patterns rather than overfitting to the specific characteristics of a single dataset. PrimeVul’s rigorous labeling exposes the true difficulty of function-level vulnerability detection, and we therefore regard the PrimeVul results as the most faithful estimate of how our models would behave in practice.

5.3. RQ3: Cross-dataset transfer

Research Question: What is the performance degradation when models trained on one dataset are evaluated on another, and which architectural choices minimize this degradation?

Table 15 presents cross-dataset transfer results using the EdgeType-GNN architecture with GraphCodeBERT embeddings. Table 16 compares how different architectures handle cross-dataset transfer (training on Draper, testing on Reveal).

Answer to RQ3: Cross-dataset transfer results in **substantial performance degradation**, with an average F1 drop of 14–37 points depending on source dataset. Models trained on Devign (nearly balanced, 45.8% vulnerable) suffer the largest transfer penalty (−37.13 avg), likely because they overfit to the balanced distribution.

Architecturally, **MultiView GGNN exhibits the best transfer retention** (78.5%), suggesting that multi-perspective aggregation generalizes better than single-view approaches. The schema heterogeneity (15 vs. 16 node types, 18 vs. 19 edge types) compounds with distribution shift to create a challenging transfer scenario.

Table 16

RQ3: Architecture impact on transfer (Draper → Reveal).

Architecture	In-Domain	Transfer	Retention
MultiView GGNN	48.67	38.21	78.5%
RGAT	49.68	36.84	74.2%
EdgeType-GNN	53.65	35.42	66.0%
Call-Aware	52.92	32.17	60.8%

Table 17

RQ4: SemVul performance across diverse benchmarks.

Dataset	Vuln%	Best F1	vs. Base	vs. Prior
Devign	45.8	66.01	+14.64	Competitive
Draper	6.2	53.65	+25.20	+0.45 (SOTA)
Reveal	9.4	48.60	+23.77	+3.33 (vs. VulBERTa)
PrimeVul	3.5	23.52	+10.68	+2.09 (vs. LM)

5.4. RQ4: Multi-dataset generalization

Research Question: Does SemVul maintain competitive performance across datasets with diverse characteristics, unlike prior approaches that excel on one benchmark but degrade on others?

A critical limitation of existing vulnerability detection methods is their tendency to achieve strong results on a specific benchmark while performing poorly on others. This raises concerns about overfitting to dataset-specific artifacts rather than learning generalizable vulnerability patterns. We evaluate SemVul’s performance consistency across four diverse benchmarks, summarized in Table 17. On a **balanced dataset (Devign, 45.8% vuln)**, SemVul achieves 66.01 F1, a +14.64 improvement over structural-only baselines. While below the originally reported Devign score (73.26 F1), this gap is consistent with documented reproducibility issues (Chakraborty et al., 2021). On two **moderately imbalanced datasets (Draper 6.2%, Reveal 9.4%)**, SemVul achieves **state-of-the-art** results on Draper (53.65 F1, +0.45 over the RNN baseline) and competitive results on Reveal (48.60 F1, +3.33 over VulBERTa, though below the 53.09 F1 reported by Vul-LMGNNs under a possibly different split). Finally, on a **highly imbalanced dataset with clean labels (PrimeVul, 3.5%)**, SemVul achieves 23.52 F1, outperforming basic fine-tuning code LM baselines (18.05–21.43 F1) by 2.09–5.47 points, demonstrating that our GNN approach competes with LLMs on rigorous benchmarks. Table 18 reports the comparison with existing state of the art models.

Answer to RQ4: Unlike prior approaches that report strong results on a single benchmark, **SemVul maintains competitive or state-of-the-art performance across all four datasets** spanning different class distributions (3.5%–45.8% vulnerable), label quality levels (noisy to rigorously verified), and graph sizes. This consistency demonstrates that combining CPG structural information with pre-trained semantic embeddings produces *generalizable* vulnerability representations rather than dataset-specific artifacts. The semantic enhancement (RQ1) appears to be the key factor: pre-trained embeddings encode universal code understanding that transfers across benchmark characteristics.

5.5. Analysis of architectural choices

We examine two key architectural choices. Regarding the **impact of GNN Depth**, Table 19 shows that shallow networks (2 layers) outperform deeper ones, aligning with the over-smoothing phenomenon where node representations converge as depth increases. As for the choice of **pooling strategy**, Table 20 shows that multi-statistic pooling (mean + max + std) captures richer distributional information, outperforming single-statistic approaches by more than 2.5 F1 points.

5.6. Comparison with prior work

Table 18 compares SemVul against published baselines across all four datasets, and more recent studies from the last two years. Comparing our approach against the published baselines, we provide the following observations:

1. Draper: SemVul achieves 53.65 F1, outperforming the best published baseline (RNN, 53.20 F1) by 0.45 points and the graph-based GGNN baseline (50.80 F1) by 2.85 points.
2. Reveal: SemVul achieves 48.60 F1, outperforming VulBERTa (45.27 F1) by 3.33 points and the original Devign model (39.55 F1) by 9.05 points.
3. Devign: our 66.01 F1 falls below the originally reported 73.26 F1, consistent with reproducibility concerns documented in the literature (Chakraborty et al., 2021).
4. PrimeVul: SemVul achieves 23.52 F1, outperforming all basic fine-tuning code LM baselines (18.05–21.43 F1), demonstrating that GNN-based approaches can compete with large language models on high-quality benchmarks.

Among the more recent studies published in the last 2 years, two of them report results on the same benchmarks we use, which enables a direct positioning of our work. GRACE (Lu et al., 2024) evaluates on Devign and Reveal, reporting an F1 of 65.11 (accuracy 59.78) on Devign and 43.13 (accuracy 89.73) on Reveal, while Vul-LMGNNs (Liu et al., 2025) reports, for its best variants, an F1 of 63.82 (accuracy 66.77) on Devign and 53.09 (accuracy 91.68) on Reveal. On Devign our best configuration achieves the highest F1 (66.01), exceeding both approaches, and on Reveal it surpasses GRACE (48.60 vs. 43.13) while trailing Vul-LMGNNs. The reported performances refers to those available in the respective published papers, since training and test splits may differ from ours. The remaining studies are not directly comparable. The adaptive-GNN work (Liang et al., 2024) evaluates only on Big-Vul (F1 of 82.9), the systematic review (Shiri Harzevili et al., 2024) reports no experiments of its own, Vul-LMGNNs uses a rebalanced version of Draper (best F1 of 85.83), which is not comparable to our original imbalanced distribution, and LLMxCPG (Lekssays et al., 2025) evaluates PrimeVul on a balanced paired test set (F1 of 62.06 with perfect precision) rather than the original imbalanced protocol followed in Table 14, so we note these protocol differences here rather than mixing incompatible settings in one table.

5.7. Computational cost

The runtime of our pipeline is best understood as a large fixed cost per function plus a smaller cost that scales with how complex that function actually is. For CPG generation, the fixed part comes from Joern itself because both the parse and the export step launch a fresh Java Virtual Machine, every function pays a start-up penalty of roughly 3.8 s regardless of its size. Table 21 makes this concrete. Here, complexity is measured not by source length but by the size of the resulting graph, namely its number of nodes (V) and edges (E), a one-line function may expand into only a handful of nodes, whereas a large control-heavy function can produce thousands.

Notice that an essentially empty function still takes 4.74 s, while a 7047-line function needs only 5.48 s, confirming that graph complexity, not line count, is what actually drives the cost. To quantify this we swept 247 functions across the full size range, summarized in Table 22, and shown graphically in Fig. 5. The time grows as $T \approx 3.78 \text{ s} + 0.284 \text{ ms} \times V$, so each additional 1000 nodes adds only about a quarter of a second. Crucially the growth is sub-linear (a log-log exponent of 0.66): a 790× increase in graph size raised the time by only about 2.3×, and the largest graph we encountered (14,229 nodes, 99,704 edges) was still generated in 7.55 s rather than minutes.

The table summarizes a least-squares fit of per-function CPG generation time against graph size, computed over a sample of $n = 247$

Table 18

Comparison with prior work, including recent 2024–2025 approaches that evaluate on our benchmarks. SemVul achieves state-of-the-art F1 on Draper, the best F1 among directly comparable approaches on Devign, competitive performance on Reveal, and outperforms basic fine-tuning code LM baselines on PrimeVul. For Devign, we include the originally reported score to highlight a known reproducibility gap in the literature.

Method	Dataset	Acc (%)	F1 (%)	ROC-AUC (%)	MCC (%)
Devign					
Devign (reported) (Zhou et al., 2019)	Devign	72.26	73.26	–	–
Vul-LMGNNs ^a (Liu et al., 2025)	Devign	66.77	63.82	–	–
GRACE ^a (Lu et al., 2024)	Devign	59.78	65.11	–	–
SemVul (this work)	Devign	59.52	66.01	68.34	–
Draper					
BOW+RF (Russell et al., 2018)	Draper	–	49.80	88.30	46.20
RNN (Russell et al., 2018)	Draper	–	53.20	89.60	50.10
BiLSTM (Zhuang et al., 2021)	Draper	92.04	49.35	–	46.10
CNN (Zhuang et al., 2021)	Draper	92.26	49.40	–	46.00
GGNN (Zhuang et al., 2021)	Draper	93.49	50.80	–	47.40
SemVul (this work)	Draper	92.77	53.65	87.28	–
Reveal					
Baseline-BiLSTM (Tian et al., 2024)	Reveal	77.13	39.11	–	–
Baseline-TextCNN (Tian et al., 2024)	Reveal	73.22	37.42	–	–
REVEAL (Chakraborty et al., 2021)	Reveal	84.37	41.25	–	–
Devign (Zhou et al., 2019)	Reveal	80.71	39.55	–	–
VulBERTa (Hanif and Maffei, 2022)	Reveal	84.48	45.27	–	–
GRACE ^a (Lu et al., 2024)	Reveal	89.73	43.13	–	–
Vul-LMGNNs ^a (Liu et al., 2025)	Reveal	91.68	53.09	–	–
SemVul (this work)	Reveal	90.04	48.60	84.53	–
PrimeVul					
CodeT5 (fine-tuned) (Ding et al., 2024)	PrimeVul	–	19.70	–	–
CodeBERT (fine-tuned) (Ding et al., 2024)	PrimeVul	–	20.86	–	–
UnixCoder (fine-tuned) (Ding et al., 2024)	PrimeVul	–	21.43	–	–
StarCoder2 (fine-tuned) (Ding et al., 2024)	PrimeVul	–	18.05	–	–
CodeGen2.5 (fine-tuned) (Ding et al., 2024)	PrimeVul	–	19.61	–	–
SemVul (this work)	PrimeVul	96.60	23.52	81.66	–

^a Results as reported in the original papers; best variant per dataset for Vul-LMGNNs; training and test splits may differ from ours.

Table 19

Impact of GNN layers (EdgeType-GNN on Devign).

Layers	F1	AUC
1	63.42	65.18
2	66.01	68.34
4	64.87	66.92
6	62.31	64.45

Table 20

Impact of pooling strategies (EdgeType-GNN on Devign)

Pooling	F1	AUC	Output Dim
Mean	62.84	65.12	<i>d</i>
Max	63.21	65.78	<i>d</i>
Mean + Max	64.57	66.89	<i>2d</i>
Multi-Stat	66.01	68.34	<i>3d</i>
Attention	65.23	67.42	<i>d</i>

Table 21

Single-function CPG generation time on Devign (median of 5 repeats).

Function	Lines	<i>V</i>	<i>E</i>	Parse (s)	Export (s)	Total (s)
(almost empty)	1	28	74	3.36	1.38	4.74
(small)	5	62	220	2.31	0.97	3.28
(largest)	7047	3190	15,138	4.05	1.43	5.48

functions that span the full range of the Devign dataset. Each row is defined below.

***n* = 247 functions.** The number of functions actually measured. We sampled across small, medium, and large functions so the fit is not biased toward any one size.

Table 22

CPG generation scaling on Devign (*n* = 247 functions).

Quantity	Value
Fixed overhead (intercept)	about 3.78 s
Marginal cost per node	0.284 ms
Linear-fit R^2 (nodes/edges)	0.78/0.81
Log-log scaling exponent	0.66 (sub-linear)
Spearman ρ (nodes vs. time)	0.84
Spearman ρ (lines vs. time)	0.71
Node range/time range	18 to 14,229/3.30 to 7.62 s
Largest graph ($V = 14,229$, $E = 99,704$)	7.55 s

***V* (nodes) and *E* (edges).** The two measures of graph complexity. *V* is the number of nodes in the generated CPG (statements, identifiers, calls, and so on) and *E* is the number of edges connecting them (control-flow, data-flow, and syntactic relations). A larger, more complex function yields a larger *V* and *E*.

***T* (time).** The wall-clock time, in seconds, to generate one CPG, counting both the joern-parse and joern-export steps.

Fixed overhead (intercept), about 3.78 s. The constant term of the linear fit $T \approx \text{intercept} + \text{slope} \times V$. It is the time a function takes even when its graph is essentially empty, and it comes almost entirely from Joern starting a fresh Java Virtual Machine for each of its two steps. This value does not depend on function size.

Marginal cost per node, 0.284 ms. The slope of the same fit, that is the extra time added for each additional node in the graph. It is very small: roughly a quarter of a second for every 1000 nodes.

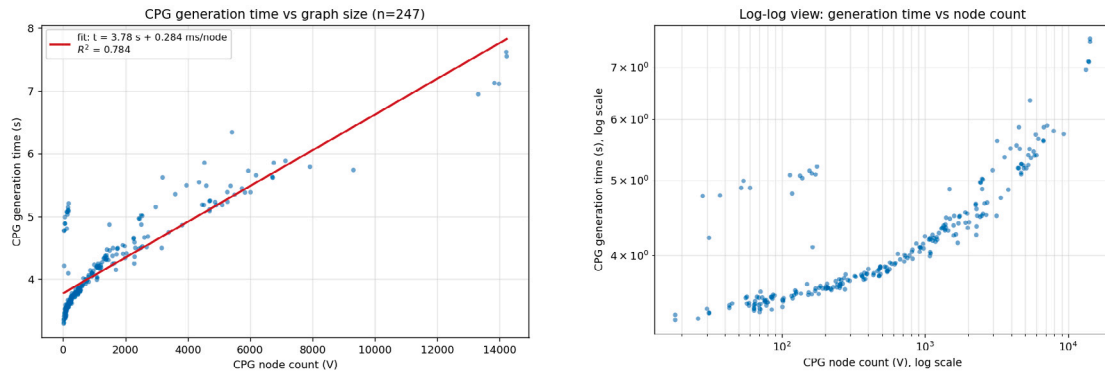


Fig. 5. CPG generation time versus graph size on Devign ($n = 247$ functions). Left: linear scale with the fitted line $T \approx 3.78 \text{ s} + 0.284 \text{ ms} \times |V|$ ($R^2 = 0.78$); the intercept is the fixed Joern/JVM overhead and the cluster of small functions sits on that floor. Right: the same data on logarithmic axes; the flat region at small node counts is the overhead-dominated regime, and the gentle slope beyond it reflects the sub-linear (exponent ≈ 0.66) growth of the complexity term.

Linear-fit R^2 (nodes/edges), 0.78/0.81. The coefficient of determination, a goodness-of-fit score between 0 and 1. A value of 0.78 means that graph size (node count) explains about 78% of the variation in generation time; using edge count instead explains about 81%. Values this high indicate that graph size is a strong predictor of time.

Log-log scaling exponent, 0.66 (sub-linear). The slope obtained when both time and node count are plotted on logarithmic axes. It captures how fast time grows with size: an exponent of 1 would mean strictly proportional growth, above 1 would mean faster-than-proportional (super-linear), and below 1 means slower-than-proportional (sub-linear). Our value of 0.66 is comfortably below 1, so doubling the graph size less than doubles the time.

Spearman ρ (nodes vs. time), 0.84. Spearman's rank correlation coefficient, a value between -1 and 1 that measures how consistently one quantity increases with another. A value of 0.84 shows a strong, reliable tendency for generation time to rise as node count rises.

Spearman ρ (lines vs. time), 0.71. The same correlation, but measured against source-code line count instead of node count. It is lower (0.71 versus 0.84), which confirms that graph complexity is a better predictor of cost than raw source length.

Node range/time range, 18 to 14,229/3.30 to 7.62 s. The smallest and largest values seen in the sample. The graphs ranged from 18 nodes up to 14,229 nodes, yet the corresponding generation time ranged only from 3.30 s to 7.62 s, showing how weakly time responds to size.

Largest graph ($V = 14,229$, $E = 99,704$), 7.55 s. The single biggest CPG in the sample, reported separately to make the point explicit: even a graph with over 14,000 nodes and nearly 100,000 edges was generated in about seven and a half seconds.

Finally, we measured how long the trained model takes to classify a single unseen function, walking up the complexity scale from the smallest function in our measured sample to the largest. All three are real Devign functions; graphs were built at exactly their measured node/edge counts and passed through the trained architecture one at a time (batch size 1) on GPU (Table 23).

Testing behaves the same way once the model is trained. Walking up the complexity scale, the smallest function in our sample, a one-line arithmetic operation whose graph has just 18 nodes, is classified in 1.60 ms; the medium function, a realistic 70-line function sitting at the dataset median (487 nodes, about 3000 edges), needs only 1.95 ms

Table 23

Per-function testing time on Devign. End-to-end is CPG generation plus inference.

Function	Lines	V	E	Inference	CPG gen.	End-to-end
Smallest	1	18	42	1.60 ms	3.30 s	3.30 s
Medium	70	487	3018	1.95 ms	3.87 s	3.87 s
Largest	3390	14,229	99,704	16.48 ms	7.55 s	7.57 s

Table 24

GNN training and testing cost on Devign.

Metric	Value
Per epoch (21,854 graphs)	about 29 s (train ~ 27.4 , val ~ 1.6)
Throughput	about 750 graphs/s
Full run (~ 80 epochs)	about 39 minutes
Test, smallest function ($V = 18$, $E = 42$)	1.60 ms
Test, medium function ($V = 487$, $E = 3018$)	1.95 ms
Test, largest function ($V = 14,229$, $E = 99,704$)	16.48 ms
End-to-end test of a new function	3.3 to 7.6 s

despite having 70 $\times$ the lines and 72 $\times$ the edges, because at this scale the GPU's fixed kernel-launch overhead of roughly 1.6 ms dominates, much like the JVM start-up does for Joern; and even the largest graph we encountered (14,229 nodes, 99,704 edges, about 33 $\times$ the edges of the medium one) takes just 16.48 ms, a factor of 8.5 rather than 33, so the growth is once again sub-linear. In other words, testing time depends somewhat on how long a function is, but far more directly on how complex its graph turns out to be, and it is never the bottleneck. Even in the worst case inference stays below 17 ms, around 0.2% of the end-to-end testing time, with the remainder being the one-off CPG generation for that new function (3.3 to 7.6 s).

The other two stages are similarly well-behaved. Embedding is precomputed once, and the related cost depends on the number of unique code strings rather than the dataset size, since identical strings are embedded once and cached; at the roughly 4600 embeddings per second we measured on GPU, even hundreds of thousands of unique strings take a couple of minutes and are then reused across every training run. Training itself is governed by the number of graphs, their average size, and the model configuration (hidden dimension, layers, and edge types), coming to about 29 s per epoch and roughly 39 min for a full run on Devign. Hardware naturally plays a role throughout: the CPG stage is CPU and JVM bound and is therefore the slowest per item, while embedding, training, and inference all run on the GPU, where they are substantially faster than on CPU. Taken together, none of the stages scales badly with function complexity, a brand-new, never-seen function goes from raw source to vulnerability verdict in under 8 s even in the worst case, and the dominant cost factor throughout is simply the number of functions in the dataset, which is exactly the behavior

Table 25
Summary of research question answers.

RQ	Key finding
RQ1	Structural-only representation (one-hot node types + edge types) achieves near-random F1 (51.37% on Devign, 12.84% on PrimeVul). Pre-trained embeddings improve F1 by +10-25 points across all datasets.
RQ2	Edge-type-aware architectures consistently outperform edge-agnostic baselines (+6.87 F1 on Draper). The optimal architecture varies by dataset: EdgeType-GNN for balanced datasets, Hypergraph/MultiView for imbalanced ones.
RQ3	Cross-dataset transfer incurs 14-37 F1 point degradation . MultiView GGNN achieves best transfer retention (78.5%). Schema heterogeneity and distribution shift compound to create challenging transfer scenarios.
RQ4	Unlike prior methods that excel on one benchmark, SemVul achieves competitive or SOTA results across all four datasets (Devign, Draper, Reveal, PrimeVul) with diverse characteristics (3.5%–45.8% vulnerable), demonstrating generalizable vulnerability detection.

one wants from a model intended for real-world use such as continuous integration or code review (see Table 24).

5.8. Summary of findings

Table 25 summarizes our answers to the four research questions. These findings suggest prioritizing embeddings over architecture complexity, as pre-trained code models provide larger gains than architectural innovations. Shallow networks (2 layers) outperform deeper models while reducing training cost. Architecture should be matched to dataset characteristics: MultiView GGNN handles class imbalance better, while EdgeType-GNN excels on balanced data. Finally, transfer penalties should be expected; practitioners should deploy models on data similar to the training distribution or employ domain adaptation techniques.

6. Conclusion

We presented SemVul, a vulnerability detection approach that combines Code Property Graphs with pre-trained code embeddings. Our evaluation across four datasets and six GNN architectures reveals that graph topology alone yields near-random performance, while semantic embeddings provide +10-25 F1 points improvement. Edge-type-aware architectures consistently outperform edge-agnostic baselines, and cross-dataset transfer incurs 14–37 F1 point penalties, with MultiView architectures exhibiting better retention. SemVul achieves state-of-the-art results on Draper (53.65 F1), competitive results on Reveal (48.60 F1), and outperforms code LM baselines on PrimeVul (23.52 vs. 18.05–21.43 F1). This consistent performance across diverse benchmarks from balanced Devign (66.01 F1) to highly imbalanced PrimeVul validates our core hypothesis: effective vulnerability detection requires both structural understanding from CPG topology and semantic understanding from pre-trained embeddings. Our reproducible methodology provides a foundation for future work on cross-language transfer, interprocedural analysis, and real-world deployment.

6.1. Interpretability and practical use

In its present form, SemVul is intended as a detection aid rather than a fully autonomous decision-maker. The model flags candidate vulnerable functions, narrowing the portion of a codebase that a security analyst must examine, while the final judgment on whether a flagged fragment constitutes an exploitable vulnerability remains with the human expert. This positioning is deliberate: given the difficulty of the task on rigorously labeled benchmarks (e.g., the low F1 scores observed on PrimeVul), automated predictions are best treated as prioritization signals that focus expert effort rather than as conclusive verdicts. A promising direction for increasing the practical value of the approach is explainability. Unlike token-based or LLM-based detectors, whose reasoning is difficult to localize in the source code, the CPG-based

representation used by SemVul is inherently structural: a prediction can, in principle, be attributed to specific nodes and edges of the graph, such as the data-flow chain linking an untrusted input to a dangerous library call, or a control-flow region lacking a protective conditional. In future work, we plan to incorporate explanation mechanisms into the pipeline — for instance, by exploiting the attention weights of edge-type-aware architectures or by extracting the subgraphs most responsible for a given prediction — so that SemVul can not only detect a vulnerability but also indicate the code elements that motivated its decision.

CRedit authorship contribution statement

Affan Younas: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Leonardo Regano:** Writing – review & editing, Visualization, Validation, Supervision, Project administration, Methodology, Formal analysis. **Giorgio Giacinto:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Funding acquisition, Conceptualization.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work the authors used OpenAI ChatGPT and Superhuman Grammarly to support grammar and language checking, and to assist with rephrasing and improving the clarity and readability of the manuscript. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

Funding

This work was partially supported by project SERICS (PE00000014) under the Italian NRRP MUR programme funded by the European Union - NextGenerationEU.

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.

Appendix. CPG schema details

This appendix provides the complete node and edge type vocabularies discovered from each dataset's CPG exports. In particular, Table 26 describes the CPG node types and their roles in vulnerability detection, while Table 27 describes the CPG edge types organized by their source graph representation.

Table 26
CPG node types and their vulnerability relevance.

Node type	Category	Datasets	Description and vulnerability relevance
CALL	Invocation	All	Function calls; most vulnerability-critical (dangerous APIs like <code>strcpy</code> , <code>sprintf</code>)
IDENTIFIER	Reference	All	Variable references; track data flow from tainted sources
LITERAL	Constant	All	Constant values including buffer sizes
BLOCK	Structure	All	Code block containers
METHOD	Entry	All	Function entry points defining analysis scope
LOCAL	Declaration	All	Local variable declarations including buffer allocations
CONTROL_STRUCTURE	Control	All	Conditionals and loops; presence determines safety checks
RETURN	Exit	All	Function exit points; relevant for resource leak detection
FIELD_IDENTIFIER	Reference	All	Struct/class field references
TYPE_DECL	Declaration	All	Type declarations
JUMP_TARGET	Control	All	Labels and jump destinations
METHOD_REF	Reference	All	References to methods/functions
UNKNOWN	Other	All	Unclassified nodes
DEPENDENCY	Import	All	External dependencies
IMPORT	Import	All	Import statements
MEMBER	Reference	Reveal, Draper	Struct/class member access patterns

Table 27
CPG edge types and their vulnerability relevance.

Edge type	Source graph	Datasets	Description and vulnerability relevance
<i>Data-Flow Edges (G_{DFG})</i>			
REACHING_DEF	DFG	All	Variable definition reaches use; critical for taint tracking
<i>Control-Flow Edges (G_{CFG})</i>			
CFG	CFG	All	Execution order; essential for detecting missing checks
DOMINATE	CFG	All	Node dominates another in control flow
POST_DOMINATE	CFG	All	Node post-dominates another
<i>Syntax Edges (G_{AST})</i>			
AST	AST	All	Parent-child syntax relationships
<i>Program Dependence Edges (G_{PDG})</i>			
CDG	PDG	All	Control dependence; conditional execution
<i>Call and Reference Edges</i>			
CALL	Call Graph	All	Function invocation relationship
ARGUMENT	Call Graph	All	Argument passed to function
REF	Reference	All	Reference relationship
PARAMETER_LINK	Call Graph	All	Parameter binding
RECEIVER	Call Graph	All	Method receiver object
<i>Type and Binding Edges</i>			
EVAL_TYPE	Type	All	Expression type evaluation
BINDS	Binding	All	Variable binding
ALIAS_OF	Alias	All	Aliasing relationship
<i>Structural Edges</i>			
CONDITION	Control	All	Condition expression
CONTAINS	Structure	All	Containment relationship
SOURCE_FILE	Structure	All	Source file association
IMPORTS	Import	All	Import relationship
CAPTURE	Closure	Reveal, Draper	Variable capture in closures

Data availability

All code, scripts, and related documentation necessary to reproduce our pipeline are available at:

<https://github.com/Affan47/SemVul>

In particular, the README.md file in the main folder of the repository illustrates the steps necessary to reproduce the experimental results detailed in this paper.

References

- Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: learning distributed representations of code. 3 (POPL), <http://dx.doi.org/10.1145/3290353>.
- Bradley, A.P., 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit. 30 (7), 1145–1159.
- Chakraborty, P., Arumugam, K.K., Alfadel, M., Nagappan, M., McIntosh, S., 2024. Revisiting the performance of deep learning-based vulnerability detection on realistic datasets. IEEE Trans. Softw. Eng. 50 (8), 2163–2177.
- Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9), 3280–3296.
- Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–33.
- Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y., 2024. Vulnerability detection with code language models: How far are we? arXiv preprint [arXiv:2403.18624](https://arxiv.org/abs/2403.18624).
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547.
- Fey, M., Lenssen, J.E., 2019. Fast graph representation learning with PyTorch Geometric. arXiv preprint [arXiv:1903.02428](https://arxiv.org/abs/1903.02428).
- Fu, M., Tantithamthavorn, C., 2022. LineVul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.
- Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E., 2017. Neural message passing for quantum chemistry. In: International Conference on Machine Learning. Pmlr, pp. 1263–1272.
- Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. UniXcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 7212–7225.

- Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.
- Hamilton, W., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. *Adv. Neural Inf. Process. Syst.* 30.
- Hanif, H., Maffei, S., 2022. VulBERTa: Simplified source code pre-training for vulnerability detection. In: 2022 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–8.
- Ji, C., Yang, S., Sun, H., Zhang, Y., 2024. Applying contrastive learning to code vulnerability type classification. In: Al-Onaizan, Y., Bansal, M., Chen, Y.-N. (Eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Miami, Florida, USA, pp. 11942–11952. <http://dx.doi.org/10.18653/v1/2024.emnlp-main.666>, URL: <https://aclanthology.org/2024.emnlp-main.666/>.
- Khosla, P., Teterwak, P., Wang, C., Sarna, A., Tian, Y., Isola, P., Maschiot, A., Liu, C., Krishnan, D., 2020. Supervised contrastive learning. *Adv. Neural Inf. Process. Syst.* 33, 18661–18673.
- Kipf, T.N., Welling, M., 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.
- Lee, J., Lee, I., Kang, J., 2019. Self-attention graph pooling. In: *International Conference on Machine Learning*. pmlr, pp. 3734–3743.
- Lekssays, A., Mouhcine, H., Tran, K., Yu, T., Khalil, I., 2025. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 489–507.
- Li, Q., Han, Z., Wu, X.-M., 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. vol. 32, (1).
- Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.
- Li, Y., Wang, S., Nguyen, T.N., 2021. Vulnerability detection with fine-grained interpretations. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. pp. 292–303.
- Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19 (4), 2244–2258.
- Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.
- Liang, C., Wei, Q., Jiang, Z., Wang, Y., Du, J., 2024. A source code vulnerability detection method based on adaptive graph neural networks. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. pp. 187–196.
- Liu, R., Wang, Y., Xu, H., Sun, J., Zhang, F., Li, P., Guo, Z., 2025. Vul-LMGNNs: Fusing language models and online-distilled graph neural networks for code vulnerability detection. *Inf. Fusion* 115, 102748.
- Lu, G., Ju, X., Chen, X., Pei, W., Cai, Z., 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *J. Syst. Softw.* 212, 112031.
- Marjanov, T., Pashchenko, I., Massacci, F., 2022. Machine learning for source code vulnerability detection: What works and what isn't there yet. *IEEE Secur. Priv.* 20 (5), 60–76.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (Eds.), *Advances in Neural Information Processing Systems*. 26, Curran Associates, Inc., URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.
- Nguyen, H.Q., Hoang, T., Dam, H.K., Ghose, A., 2025. Graph-based explainable vulnerability prediction. *Inf. Softw. Technol.* 177, 107566.
- Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 757–762.
- Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., Welling, M., 2018. Modeling relational data with graph convolutional networks. In: *European Semantic Web Conference*. Springer, pp. 593–607.
- Shiri Harzevili, N., Boaye Belle, A., Wang, J., Wang, S., Jiang, Z.M., Nagappan, N., 2024. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Comput. Surv.* 57 (3), 1–36. <http://dx.doi.org/10.1145/3699711>.
- Sokolova, M., Lapalme, G., 2009. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.* 45 (4), 427–437.
- Song, Z., Wang, J., Liu, S., Fang, Z., Yang, K., 2022. HGvul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. *Secur. Commun. Netw.* 2022 (1), 1919907.
- Tian, Z., Tian, B., Lv, J., Chen, Y., Chen, L., 2024. Enhancing vulnerability detection via AST decomposition and neural sub-tree encoding. *Expert Syst. Appl.* 238, 121865.
- Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A., Stringhini, G., 2024. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In: 2024 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 862–880.
- Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks.
- Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., Hoi, S., 2023. CodeT5+: Open code large language models for code understanding and generation. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. pp. 1069–1088.
- Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In: *Proceedings of the 45th International Conference on Software Engineering, ICSE '23*, IEEE Press, ISBN: 9781665457019, pp. 2275–2286. <http://dx.doi.org/10.1109/ICSE48619.2023.00191>.
- Xu, K., Hu, W., Leskovec, J., Jegelka, S., 2018. How powerful are graph neural networks?
- Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.
- Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., Leskovec, J., 2018. Hierarchical graph representation learning with differentiable pooling. *Adv. Neural Inf. Process. Syst.* 31.
- Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* 32.
- Zhuang, Y., Suneja, S., Thost, V., Domeniconi, G., Morari, A., Laredo, J., 2021. Software vulnerability detection via deep learning over disaggregated code graph representation. *arXiv preprint arXiv:2109.03341*.