DOI: https://doi.org/10.71317/kjard.2.11.2026.1093

Kashmir Journal of Academic Research and Development

Journal homepage: https://rjsaonline.org/index.php/KJARD

Graph Neural Networks for Interprocedural Vulnerability Detection Using Software Dependency and Program-Flow Graphs

Talha Hussain
Faculty of Computing, Riphah International University, Islamabad 46000, Pakistan
Email: thussain98ml@gmail.com
Farhan Bashir Shaikh
Assistant Professor, Department of Computer Science, The University of Larkano
Email: Farhan@uolrk.edu.pk

ARTICLE INFO

Received:
October 19, 2026
Revised:
October 31, 2026
Accepted:
November 11, 2026
Available Online:
November 21, 2026

Keywords:
Graph Neural Networks, Vulnerability Detection, Interprocedural Analysis, Code Property Graph, System Dependence Graph, Program Slicing,Taint Analysis, Static Analysis, Deep Learning, Software Security

Corresponding Author:
Talha Hussain
Email:
thussain98ml@gmail.com

ABSTRACT

Software vulnerabilities remain a critical threat to modern computing infrastructure, with reported flaws increasing annually and attack surfaces expanding rapidly through open-source ecosystems. Traditional detection approaches static rule-based analyzers, dynamic fuzzing, and symbolic execution suffer from high false-positive rates, path explosion, and poor scalability. Early machine learning methods treating code as flat token sequences fail to capture structural and semantic dependencies, experiencing severe performance degradation on realistic benchmarks. Graph Neural Networks (GNNs) have emerged as a powerful paradigm, modeling programs as structured graphs including Abstract Syntax Trees, Control Flow Graphs, Data Flow Graphs, and Program Dependence Graphs. However, most existing GNN-based detectors operate intraprocedurally, missing cross-function vulnerabilities such as buffer overflows and use-after-free that depend on interprocedural data flows. This review provides a comprehensive evaluation of GNN architectures, software dependency graph representations, and program-flow models for interprocedural vulnerability detection. We examine theoretical foundations of Code Property Graphs and System Dependence Graphs, analyze advanced architectures including relational, hyperbolic, quantum-enhanced, and hybrid language-graph models, and explore interprocedural flow mechanics incorporating program slicing, taint analysis, and executable vulnerability chains. We present comparative analyses of state-of-the-art detectors, evaluate benchmark datasets and label quality issues, and investigate explainability, causal graph learning, and adversarial robustness. Finally, we identify open challenges including graph construction scalability, long-range dependency modeling, LLM integration, and automated patch generation, synthesizing key conclusions to guide future research in scalable, interpretable, and robust vulnerability detection.

Introduction

Software vulnerabilities represent one of the most critical security risks facing modern computing infrastructure, serving as primary vectors for remote code execution, privilege escalation, severe data leakage, and system-wide supply-chain compromises (Perez-Botero et al., 2013). Public databases highlight an accelerating influx of security flaws; for instance, official security records demonstrate that newly reported vulnerabilities grew to 24,801 annually, representing a nearly 20% year-over-year increase, with an alarming growth in the proportion of ultra-high-risk vulnerabilities (Dong et al., 2019). Simultaneously, the rapid expansion of open-source software ecosystems exemplified by tens of millions of new repositories added annually to platforms like GitHub has exponentially expanded the attack surface. Flaws introduced in deeply embedded, low-level open-source libraries rapidly propagate across dependent software stacks, creating systemic vulnerabilities that demand scalable, automated, and mathematically rigorous static analysis frameworks (Zhou et al., 2026).

Traditional vulnerability detection mechanisms have historically relied on static analyzers, dynamic fuzzing, and symbolic execution. Static application security testing (SAST) tools rely heavily on human-engineered expert rules and pattern-matching heuristics, which suffer from prohibitively high false-positive rates and fail to capture complex, non-linear semantics (Li et al., 2018). Dynamic analyzers and symbolic execution engines provide precise execution paths but are severely bottlenecked by exponential path explosion, high computational overhead, and incomplete code coverage, rendering them impractical for continuous integration and large-scale repository scanning (Yun et al., 2018).

To overcome these constraints, early machine learning approaches adopted natural language processing (NLP) paradigms, treating source code as flattened token sequences (Xu & Zhu, 2022). While sequence models capture localized lexical patterns, they treat code as unidimensional text, ignoring the structural syntax and semantic control-data flow dependencies inherent in compiled or interpreted programs. Empirical evaluations reveal that token-based models frequently learn superficial textual correlations rather than true flaw mechanics (He et al., 2025). When synthetic artifacts and duplicate code samples are removed from evaluation benchmarks, token-based detectors experience performance collapses of over 50%, with precision dropping by up to 95% when deployed on full project repositories (Chen et al., 2022).

The fundamental limitation of sequence-based learning has driven a paradigm shift toward Graph Neural Networks (GNNs). Programs are inherently graph-structured objects defined by hierarchical abstract syntax trees (ASTs), control flow graphs (CFGs), data flow graphs (DFGs), and program dependence graphs (PDGs) (Dandibhotla et al., 2025). By unifying these structures into composite program graphs such as the Code Property Graph (CPG) GNNs can propagate semantic and topological information across node neighborhoods, capturing deep structural patterns (Liu, 2020).

However, early GNN-based vulnerability detectors predominantly evaluated isolated code fragments or single functions within an intraprocedural scope. Real-world software flaws rarely reside entirely within a single function boundary (Shuvo, 2026). Critical security weaknesses such as buffer overflows, memory corruption, use-after-free, and integer underflows frequently depend on interprocedural flows, where untrusted user input originating in one module flows through complex caller-callee hierarchies, struct field assignments, and inter-function pointer arithmetic before reaching a vulnerable operation or sink (Cao et al., 2023). Intraprocedural detectors operate on localized graph neighborhoods, failing to track these long-range data dependencies and interprocedural control transfers, which leads to high false-negative rates and uninterpretable predictions (Shuvo, 2026).

This review article provides a comprehensive evaluation of Graph Neural Network architectures, software dependency graph representations, and program-flow models specifically designed for interprocedural vulnerability detection (He et al., 2026). Section 2 examines the theoretical foundations of program graph abstractions, detailing the formal constructs of CPGs and System Dependence Graphs (SDGs). Section 3 analyzes advanced GNN architectures, ranging from multi-relational and edge-conditioned networks to hyperbolic and quantum-enhanced models (Bhatia & Arora, 2012). Section 4 explores interprocedural flow mechanics, including executable vulnerability chains, program slicing, and automated CPG traversals. Section 5 presents a comparative analysis of state-of-the-art detectors. Section 6 evaluates benchmark datasets, label quality issues, and evaluation methodologies (Gong et al., 2023). Section 7 investigates model explainability, causal graph learning, and adversarial robustness. Section 8 details open research challenges, and Section 9 synthesizes key conclusions (Wu et al., 2022).

Theoretical Foundations of Program Graph Representations

Unified Program Graphs: AST, CFG, DFG, and Call Graphs

Extracting structured semantics from source code requires decomposing program execution and syntax into distinct, formal graph representations (Borowski et al., 2024). Syntactic parsing transforms source code into an Abstract Syntax Tree (AST), represented as a rooted tree where internal nodes denote operators, control constructs, and statement types, while leaf nodes hold operators, identifiers, and literals. The AST enforces grammatical hierarchy and expression priorities but lacks execution order awareness (Kaplan et al.,1995).

To capture operational semantics, a Control Flow Graph (CFG) models the execution order of program statements, where nodes represent basic blocks or individual statements, and directed edges represent execution transfers, conditional branching, and loop iterations (Devkota & Isaacs, 2018). Operating alongside the CFG, the Data Flow Graph (DFG) and Program Dependence Graph (PDG) combine data and control dependencies. Data dependency edges indicate that a value assigned to a variable at a source statement is directly referenced or mutated at a target statement, whereas control dependency edges signify that the evaluation of a predicate statement directly conditions the execution of target statements (Naeem et al., 2019). Finally, to extend representations across procedural boundaries, the Call Graph (CG) captures interprocedural relationship networks, where nodes represent individual methods or functions, and directed edges denote call site invocations from callers to callees (Grove et al.,1997).

Code Property Graph (CPG) and System Dependence Graph (SDG)

The Code Property Graph (CPG) unifies AST, CFG, and PDG representations into a single multi-relational property graph structure. Formally, a function-level or file-level CPG is defined as a tuple:

$$G = (V, E, X_v, X_e)$$

where V represents the set of heterogeneously typed program nodes (such as method definitions, identifier references, function calls, control structures, and local variable declarations), E represents directed multi-relational edges, $X_v \in \mathbb{R}^{|V| \times d}$ denotes initial node feature embeddings, and X_e captures edge-type attribute matrices (Wei et al., 2026).

The edge vocabulary of a CPG explicitly categorizes structural and semantic relationships using distinct edge types (Chen at al., 2026) . Syntactic parent-child relations are maintained via AST edges, sequential execution flow and conditional branching are encoded through CFG edges, data reaching definitions and control dependencies are established via REACHING_DEF and CDG edges, and function invocations across scope boundaries are captured by CALL edges (Dai et al., 2025). Additional specialized relations include EVAL_TYPE, ARGUMENT, and DOMINATE edges. Structural distributions across software codebases demonstrate that syntax-focused AST edges dominate CPG topologies (~38%), followed by sequential CFG edges (~15%) and data-flow REACHING_DEF edges (~12%) (Baldwin et al., 2014).

To elevate CPG representations to the interprocedural scale, the System Dependence Graph (SDG) extends PDGs across method boundaries (Horwitz et al., 2004). An SDG constructs interprocedural call-control edges that link call sites directly to target function entry points, alongside interprocedural parameter dependence edges that link actual arguments at call sites to formal parameters in target methods, and formal return values back to call-site receivers. This structural coupling enables precise tracking of reaching definitions and taint propagation across complex interprocedural execution pathways (Qasem et al., 2024).

Mathematical Formulation of Graph-Based Vulnerability Detection

Graph-based vulnerability detection models map structured program representations into discrete classification spaces. Let a program graph G_P be defined by node feature matrix X and a set of relation-specific adjacency matrices $\{A_r\}_{r \in \mathcal{R}}$, where $A_r \in \{0,1\}^{|V| \times |V|}$ indicates adjacency under relation r.

For graph-level binary or multi-class vulnerability classification, the objective is to learn a parameterized neural function h_θ:

$$h_\theta(X, \{A_r\}_{r \in \mathcal{R}}) \to c, \quad c \in \{0,1, \dots, K\}$$

where $c = 0$ denotes a clean program, and $c > 0$ maps to specific Common Weakness Enumeration (CWE) vulnerability classes, such as CWE-119 for buffer overflows or CWE-89 for SQL injection.

For statement-level fine-grained localization, the problem is formulated as a node classification task on the program dependence graph $G_P = (V, E)$:

$$f_\theta : V \to \{0,1\}$$

where each node $v_i \in V$ corresponding to a statement line is assigned a binary probability indicating whether it constitutes a vulnerable statement or part of an exploitable propagation path (George & Thampi, 2018).

Deep Graph Neural Network Architectures for Code Security

Homogeneous and Heterogeneous Backbone Models

Graph Neural Network backbones perform iterative message passing to compute context-aware node and graph embeddings. Given a node v_i with hidden state $h_i^{(l)}$ at layer l, message aggregation is defined across neighbor sets $\mathcal{N}(i)$ (Gavette et al., 2018).

Graph Convolutional Networks (GCN) compute symmetric normalized aggregations across spatial neighborhoods:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\hat{D}_{ii} \hat{D}_{jj}}} A_{ij} W^{(l)} h_j^{(l)} \right)$$

where $\hat{A} = A + I_N$ represents the adjacency matrix with self-loops, $\hat{D}$ is the diagonal degree matrix, and $W^{(l)}$ is a trainable weight matrix.

Graph Attention Networks (GAT) incorporate dynamic attention coefficients α_{ij} to weigh neighbor interactions based on semantic importance:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)}\right)$$

$$\alpha_{ij}^{(l)} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^T\left[W^{(l)}h_i^{(l)} \,||\, W^{(l)}h_j^{(l)}\right]\right)\right)}{\sum_{k\in\mathcal{N}(i)}\exp\left(\text{LeakyReLU}\left(\mathbf{a}^T\left[W^{(l)}h_i^{(l)} \,||\, W^{(l)}h_k^{(l)}\right]\right)\right)}$$

where **a** is a learnable attention vector and || represents concatenation.

Gated Graph Neural Networks (GGNN) utilize recurrent gated units (GRUs) to update node states across T propagation timesteps, making them effective for tracking sequential execution flows across CFGs and PDGs.

To process multi-relational CPGs without collapsing edge distinctiveness, Relational Graph Convolutional Networks (RGCN) and R-GGNN assign edge-type specific weight matrices $W_r^{(l)}$ for each relation $r \in \mathcal{R}$:

$$h_i^{(l+1)} = \sigma\left(W_0^{(l)}h_i^{(l)} + \sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_r(i)}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)}\right)$$

where $\mathcal{N}_r(i)$ denotes neighbors connected via relation r, and $c_{i,r}$ is a normalization constant.

Advanced and Frontier Deep Architectures

To address structural challenges in code modeling such as edge-type dominance skew, hierarchical nesting, and non-Euclidean dependency structures advanced architectures incorporate specialized mathematical operations (Meinye & Swart, 2024).

Code syntax trees and nested block hierarchies exhibit exponential expansion properties that cause crowding and distortion when mapped into low-dimensional Euclidean spaces (Sulc, 2025). Hyperbolic Graph Neural Networks (FGH) resolve this limitation by mapping node features onto hyperbolic manifolds, such as the Poincaré ball or Lorentz model (Yang et al., 2022). Feature updates utilize Möbius gyrovector addition $\oplus^c$ and matrix-vector multiplication $\otimes^c$ in tangent space:

$$\mathbf{x}^H = \exp_{\mathbf{0}}^c(\mathbf{x}^E), \quad \mathbf{h}_i^{(\ell,H)} = \left(\mathbf{W}^{(\ell)} \otimes^c \mathbf{x}_i^{(\ell-1,H)}\right) \oplus^c \mathbf{b}^{(\ell)}$$

Hyperbolic mappings preserve syntactic tree hierarchies and nested scopes with minimal embedding distortion, significantly improving structural representation fidelity (Zhou, 2026)

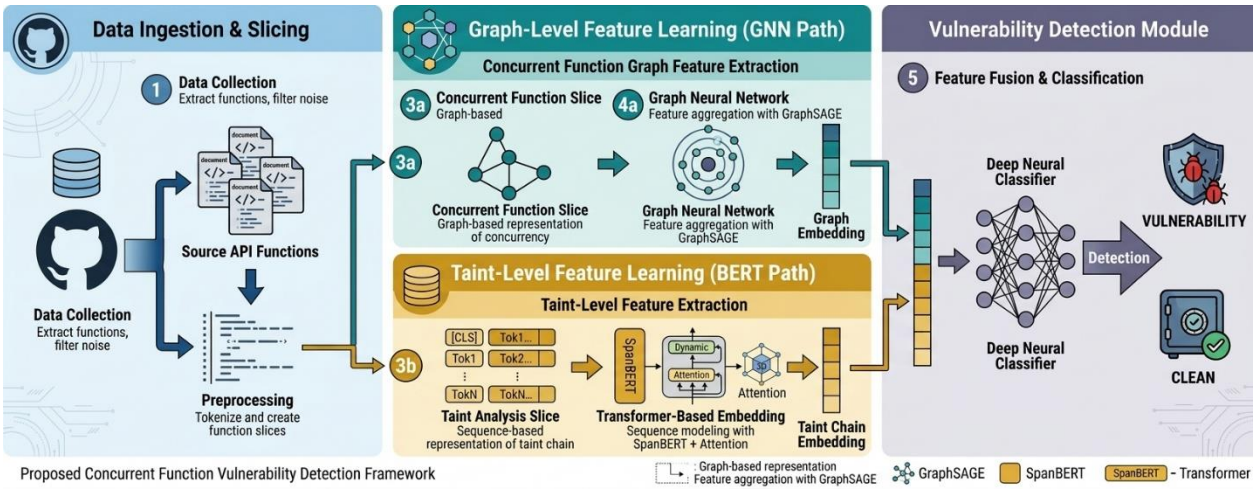
In parallel, Quantum-Enhanced GNNs (QCNN) incorporate parameterized quantum circuits (RQCs) within convolutional and self-attentive pooling layers. By combining bi-affine layers with GCN backbones, quantum-enhanced models preserve complex syntactic dependency parsing while compressing multi-scale features for high-precision classification (Rahaman & Nguyen, 2026).

Hybrid Language-Graph Models

Modern vulnerability detection frameworks increasingly combine pre-trained language models (PLMs) with GNN backbones to capture both textual semantics and structural dependencies (Zhou et al., 2026).

The LineVD framework encodes raw statement tokens using Transformer models such as CodeBERT to generate dense initial statement representations, which are subsequently passed through multi-layer GAT networks operating over PDG data and control dependency edges (Hin et al., 2022). The SemVul architecture integrates pre-trained code models with multi-relational CPG structures and multi-statistic pooling, unifying structural topology with language semantics to prevent out-of-distribution performance collapse across unseen software repositories. Similarly, hybrid architectures combining Graph CodeBERT with structural graph encodings (GCBERT + Struct) allow models to learn deep contextual representations while preserving explicit interprocedural flows (Tiftikci et al., 2026).

Figure 1: Architectural Workflow of the Proposed Concurrent Function Vulnerability Detection Framework Combining Graph-Level GNN and Taint-Level BERT Paths.



Interprocedural Analysis Frameworks and Flow Mechanics

Interprocedural Call Graph Construction and Propagation

Intraprocedural analysis treats functions as isolated units, failing to detect vulnerabilities whose root causes and sinks reside in separate methods. Interprocedural analysis bridges function-level CPGs into project-wide graph databases using static call graphs (Grove & Chambers, 2001). Static analyzers such as Joern parse method invocation nodes and connect caller functions directly to target callee declarations through interprocedural call edges. When an untrusted input enters a caller method, data flow definitions propagate through parameter mappings into callee methods, allowing models to trace memory operations across procedural boundaries until reaching a vulnerable sink (Wang et al., 2020).

Constructing accurate interprocedural call graphs requires resolving dynamic method dispatch, function pointers, callback registrations, and external library calls. Failure to resolve indirect call targets breaks call graph connectivity, resulting in truncated data flow paths and undetected security flaws (Yang et al., 2025).

Executable Vulnerability Chains and Causal Flow Graph Construction

Recent research shifts away from point-wise vulnerability predictions toward extracting executable, causal interprocedural chains. An interprocedural vulnerability chain explicitly models the ordered transition from a root cause, through a propagation path, to an exploitable sink (Shuvo, 2026).

To construct valid chains, static and dynamic dependencies are enriched with alias analysis and control-flow reachability constraints. Causal graph attention mechanisms suppress non-causal context nodes, ensuring that message propagation is restricted to semantically viable execution paths. This eliminates spurious correlations caused by benign refactoring or dead code, producing execution traces that reflect true flaw mechanics (Zhang & Barthe, 2025).

Program Slicing and Taint Analysis

Program slicing isolates the subset of a program that influences or is influenced by a specific variable at a designated program point, known as the slicing criterion (Tip, 1994). Backward slicing originates from a security-sensitive sink such as memory copy invocations, memory deallocation calls, system commands, or raw pointer arithmetic and traverses incoming data and control dependence edges backward to locate untrusted input sources. Forward slicing traces untrusted user inputs forward along outgoing dependency edges to identify reachable sensitive functions (Rao et al., 2024).

Frameworks such as DeepWukong and VulDeeLocator rely on slice-level graph abstractions. VulDeeLocator generates slices at the intermediate code representation level (such as LLVM IR) to preserve low-level memory semantics (Cheng et al., 2021). However, intermediate code slicing introduces an operational bottleneck: source code that cannot be fully compiled due to missing header dependencies or incomplete build environments cannot be processed (Yadavally et al., 2024).

Automated graph traversal languages, such as CPGQL executed over Joern databases, allow analysts to formulate programmatic interprocedural taint queries. Backward reachability traversals locate code paths where argument nodes at sensitive sinks are

reachable by data flows originating from user-controlled input sources, extracting interprocedural paths across complex software systems (Lekssays, 2026).

Comparative Analysis of State-of-the-Art Approaches

Graph-based vulnerability detection models differ significantly across target granularities, underlying graph abstractions, neural architectures, and interprocedural capabilities (Lee et al., 2018). Table 1 categorizes representative deep learning vulnerability detectors based on their technical characteristics.

Figure 2: Comprehensive Performance Evaluation Dashboard and Unified ROC Comparison Across Multi-Relational Graph Representations and Categorical Tasks.

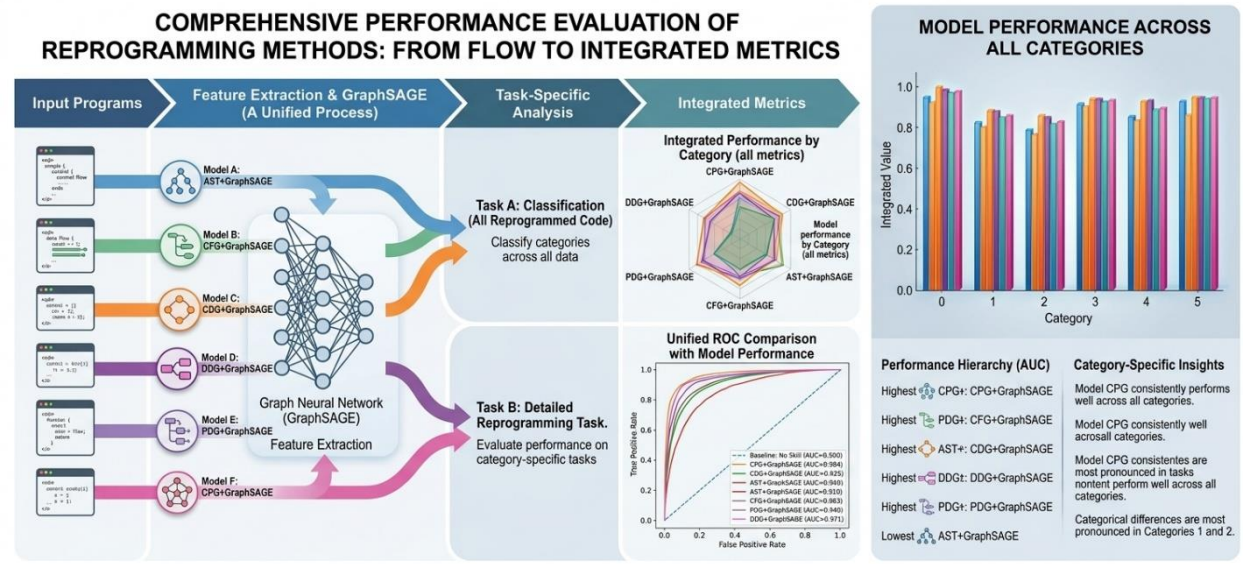


Table 1: Classification and Technical Characteristics of Representative Graph-Based Vulnerability Detection Models

Model Name	Target Granularity	Underlying Code Representation	GNN Architecture	Interprocedural Support	Primary Strengths & Limitations
Devign [cite: 1, 25]	Function-level	Composite Graph (AST + CFG + DFG + Code Sequence)	Gated Graph Recurrent Network (GGNN)	No (Intraprocedural scope only)	Fuses structural and sequential properties; high graph construction overhead and unable to detect cross-function flaws.
ReVeal	Function-level	Code Property Graph (CPG)	GGNN with imbalanced learning re-sampling	No (Limited to function scope)	Robust on realistic, imbalanced datasets; high computational overhead during CPG extraction.
DeepWukong	Slice-level	Program Dependence Graph (PDG) Slices	GCN / GAT with Doc2Vec node embeddings	Partial (Intra-file program slices)	High precision on localized slices; depends on manual slicing criteria and misses cross-file flows.
VulDeeLocator	Line/Slice-level	Intermediate Representation (LLVM IR Slices)	BiGRU + k-Max Pooling / Custom Neural Network	Partial (Interprocedural via IR call graphs)	Precise line-level localization using low-level semantics; strictly requires buildable, compilable source code.

IVDetect	Statement-level	PDG with tokenized subtrees	Graph Convolutional Network (GCN)	No (Intra-function PDGs)	Interpretable statement predictions; vulnerable to unstable interpretation outputs and high false-positive rates.
LineVD	Statement-level	PDG + Token Sequences	Dual Graph Attention Networks (GAT) + CodeBERT	No (Intraprocedural statement graphs)	Achieves +105% F1-score improvement over IVDetect; ignores interprocedural data propagation.
SemVul	Function / Project-level	CPG + Pre-trained Semantics	Edge-Aware Multi-Layer GNN + Multi-Statistic Pooling	Partial (Scalable multi-file graph structures)	Language-agnostic, robust to cross-benchmark distribution shifts; higher memory consumption.
Chain-Centric GNN	Interprocedural Chain-level	Interprocedural CPG Extensions (C1 Graphs)	GCBERT + Struct-only Causal Graph Attention	Full (Explicit caller-callee propagation chains)	Generates executable, causally consistent root-to-sink chains; requires extensive interprocedural parsing setup.
SCVHunter	Contract / Function-level	Heterogeneous Semantic Graph	Heterogeneous Graph Attention Network (HAN)	Partial (Cross-function contract calls)	Effective for multi-label smart contract flaws; vulnerable to adversarial refactoring attacks.

Table 2 compares empirical performance metrics reported across benchmark evaluation datasets, highlighting performance trade-offs across prediction granularities.

Table 2: Performance Evaluation and Benchmark Metric Comparison Across Standard Datasets

Model	Benchmark Datasets	F1-Score Range (%)	Precision (%)	Recall (%)	Vulnerability Localization Granularity
Devign	Devign, CodeXGLUE	62.4 - 71.5	64.1 - 73.2	60.8 - 70.1	Function-level classification.
ReVeal	ReVeal, Big-Vul	58.2 - 68.7	61.5 - 72.0	55.3 - 65.8	Function-level classification.
DeepWukong	SARD, Redis, Lua	64.0 - 82.3	68.2 - 85.1	60.3 - 79.8	Code slice-level localization.
VulDeeLocator	SARD, NVD	82.5 - 88.1	84.1 - 90.3	81.0 - 86.2	Fine-grained line-level localization.
IVDetect	Big-Vul, Devign	32.1 - 44.8	35.0 - 48.2	29.8 - 41.9	Statement-level node classification.
LineVD	Big-Vul, D2A	65.8 - 78.4	70.1 - 81.2	62.0 - 75.8	Statement-level node classification.
Vuldetexp	Big-Vul	88.0	89.5	86.6	Slice-level with line-level explanation.
UntrustVul (Enhanced)	Benchmark Ensemble	82.0 - 94.0	83.5 - 95.1	80.8 - 93.0	Graph classification with trust verification.
Chain-Centric GNN	ReposVul (C/C++)	74.2 - 83.6	76.8 - 86.0	71.8 - 81.5	Executable interprocedural causal chain.

Quantitative comparisons reveal a key architectural trade-off: while function-level models (such as Devign and ReVeal) demonstrate higher nominal macro-F1 scores on curated function benchmarks, their practical utility is limited because they fail to identify the exact line or interprocedural path containing the flaw (Soni, 2026). Conversely, statement-level models (such as IVDetect and LineVD) provide actionable localization for security developers, but encounter severe performance drops when handling complex control structures, requiring rich contextual pre-training (such as CodeBERT) to maintain high precision (Fu & Tantithamthavorn, 2022).

Benchmarks, Dataset Quality, and Evaluation Methodologies

Standard Dataset Ecosystem

Evaluating graph-based vulnerability detection models relies on several major benchmark datasets, each offering distinct characteristics. The Devign dataset comprises C/C++ functions collected from high-profile open-source repositories such as QEMU and FFmpeg, manually labeled by security experts (McDermott et al., 2026). The ReVeal dataset, extracted from Linux kernel and Wireshark codebases, was specifically designed to reflect realistic label class imbalances (~10% vulnerable versus ~90% non-vulnerable) (Chakraborly et al., 2024).

For large-scale evaluations, Big-Vul and MegaVul provide extensive collections scraped from historical CVE and NVD records across C/C++ commit histories. MegaVul provides explicit Joern CPG extractions containing 33 node types and 21 edge types (Ni et al., 2024). Synthetic benchmarks, such as NIST's Software Assurance Reference Dataset (SARD), provide clean, isolated vulnerability test cases, whereas datasets like D2A utilize static analysis tools to construct auto-labeled differential graph benchmarks (Black, 2018). More recently, the ReposVul dataset introduced repository-level benchmarks designed specifically for evaluating interprocedural models, capturing multi-file contexts and caller-callee networks (Wang et al., 2024).

Data Quality Crisis: Noise, Leakage, and Duplication

Empirical evaluations reveal systemic data quality issues across widely used vulnerability benchmarks (Brown et al., 2013). Synthetic datasets like SARD contain predictable code structures and uniform naming conventions. Models trained on synthetic benchmarks achieve artificially high F1-scores (>90%) but fail completely when deployed on complex open-source software (She et al., 2026).

Furthermore, historical datasets suffer from extensive data leakage and code duplication, containing cloned code fragments and overlapping commit states between training and test splits (Ding et al., 2024). Removing duplicate code via decision tree filtering and exact token matching causes baseline model performance to drop drastically, demonstrating that traditional GNNs often memorize code clones rather than learning structural vulnerability patterns (Lee at al., 2022). Finally, real-world software repositories typically exhibit extreme class imbalance, where vulnerable lines represent less than 1% of total code. Models trained on artificially balanced 50/50 splits suffer severe precision loss when evaluated under realistic low-prevalence scenarios (Roesler et al., 2026).

To mitigate these dataset pathologies, robust data preprocessing pipelines implement duplicate code removal, outlier handling via log transformations, Z-score vector normalization, and stratified temporal splitting (splitting train/test data based on commit timestamps to prevent future-to-past data leakage) (Dhawas et al., 2024).

Explainability, Causality, and Adversarial Robustness

Post-hoc Explanations and GNN Interpreters

Because deep GNN architectures operate as complex models, deploying them in security-critical environments requires interpretable, human-verifiable explanations. Interpreters such as GNNExplainer, ProvX, Vuldetexp, and SAVED isolate minimal subgraphs and critical node features that maximize the model's mutual information for a given prediction (Chen et al., 2024).

However, empirical guidelines evaluating GNN interpreters reveal substantial stability and consistency issues (Hu et al., 2023). Post-hoc interpreters frequently produce unconvincing explanations, where different interpreter runs on the same trained GNN model yield non-deterministic, conflicting node importance heatmaps, rendering explanations untrustworthy for security analysts (Kosan et al., 2024).

Counterfactual Reasoning and Faithfulness

To overcome interpreter instability, recent frameworks adopt counterfactual explanations and topological regularization. Counterfactual reasoning evaluates what minimal structural modifications such as adding a missing bound check or removing a specific pointer assignment would flip the model's prediction from vulnerable to secure (Paul et al., 2024).

Research demonstrates that explanation faithfulness alone is insufficient grounds for trusting model predictions (Papenmeier et al., 2019). Frameworks like UntrustVul evaluate prediction trustworthiness across tens of thousands of samples, detecting unreliable predictions with F1-scores between 82% and 94%. Incorporating trust verification and counterfactual feedback into closed-loop training enhances overall detection F1-scores by up to 321% (Ribeiro et al., 2016).

Adversarial Attacks and Defense Mechanisms

Graph Neural Networks are highly vulnerable to adversarial manipulation. Malicious actors or automated transformation tools can apply semantics-preserving code refactorings such as variable renaming, function wrapping, dead code insertion, and equivalent loop transformations to alter graph topology while preserving exact runtime behavior (Khamaiseh et al., 2022). Under such refactoring attacks, baseline GNN detectors experience catastrophic performance degradation, with Attack Success Rates (ASR) reaching up to 18.73% (Chumo, 2026).

To counter adversarial manipulation, robust architectures incorporate causal attention and topological regularization (Fu et al., 2026). The ORACAL framework limits performance degradation under refactoring attacks to an F1 decrease of approximately 2.35%, maintaining a low Attack Success Rate (ASR) of 3%, significantly outperforming baselines like SCVHunter and MANDO-HGT. Similarly, causality-aware graph attention filters out spurious non-causal context nodes, suppressing features altered during benign code refactorings (Shuvo, 2026).

Open Challenges and Future Directions

Despite significant advances in graph neural networks for vulnerability detection, several foundational technical challenges remain open:

Graph Construction and Scalability Bottlenecks: Generating multi-relational CPGs and SDGs for repository-scale systems containing millions of lines of code requires extensive CPU and memory overhead (Du Bois et al., 2013). Constructing full project-level property graphs can take hours, creating significant friction for real-time developer feedback in Continuous Integration / Continuous Deployment (CI/CD) pipelines. Developing lightweight, incremental graph extraction algorithms remains a critical priority (Fan & Tian, 2022).

Context Windows and Dynamic Dispatch Resolution: Modern GNN message passing is typically restricted to 2-hop or 3-hop local neighborhoods to prevent oversmoothing (Liu et al., 2026). However, interprocedural execution flows frequently stretch across dozens of method calls, dynamic class inheritances, and virtual function tables. Modeling these long-range interprocedural dependencies without losing node gradient signal requires deeper, non-oversmoothing GNN backbones and efficient memory structures (Lukobnikov & Fischer, 2021).

Synergistic Integration with Large Language Models: While GNNs excel at structural program parsing, LLMs possess vast open-domain semantic understanding (Romero et al., 2023). Modern hybrid approaches such as codebadger and LLMxCPG use LLMs as multi-agent orchestration engines that programmatically query CPG databases using CPGQL. Combining LLMs for reasoning with GNN and CPG platforms for deterministic taint tracking represents a promising research frontier (Siewruk & Berej, 2026).

Closed-Loop Remediation and Patch Generation: Beyond detecting and localizing vulnerabilities, automated systems must generate executable, context-aware security patches (Apuri et al., 2026). Formalizing vulnerability repair as graph-conditioned patch generation where models take vulnerable CPG subgraphs, localized line sets, and CWE type tags as input to synthesize verified repair patches represents the next major evolution in autonomous software security (Seo et al., 2025).

Conclusion

This review has systematically examined the landscape of Graph Neural Network-based approaches for interprocedural vulnerability detection, tracing the evolution from token-based sequence models to sophisticated graph-structured learning frameworks. The fundamental insight driving this paradigm shift is clear: programs are inherently graph-structured objects, and vulnerability detection requires capturing the complex interplay of syntax, control flow, data dependencies, and interprocedural call relationships that flat text representations fundamentally cannot encode. Our analysis reveals several critical findings. First, Code Property Graphs and System Dependence Graphs provide mathematically rigorous foundations for unifying heterogeneous program representations AST, CFG, DFG, and call graphs into composite structures amenable to neural message passing. Second, advanced GNN architectures including Relational GCNs, Gated Graph Neural Networks, hyperbolic embeddings, and hybrid language-graph models offer complementary strengths: relational models preserve edge-type semantics, hyperbolic spaces capture hierarchical nesting without distortion, and PLM-GNN hybrids combine textual semantics with structural topology. Third, interprocedural analysis frameworks leveraging program slicing, taint propagation, and executable vulnerability chains substantially outperform intraprocedural detectors, yet remain challenged by indirect call resolution and scalable graph construction. The comparative analysis of state-of-the-art detectors exposes a persistent tension between granularity and performance: function-level models achieve higher nominal F1 scores but provide limited actionable localization, while statement-level and slice-level models offer precise vulnerability identification at the cost of increased computational overhead and sensitivity to dataset quality. Benchmark evaluations reveal systemic data quality crises synthetic

dataset bias, code duplication, label noise, and unrealistic class distributions that inflate reported performance and obscure true generalization capabilities. Looking forward, several research frontiers demand attention. Scalable, incremental graph extraction algorithms are essential for real-time CI/CD deployment. Non-oversmoothing GNN backbones capable of modeling long-range interprocedural dependencies beyond 2-3 hop neighborhoods remain an open architectural challenge. The synergistic integration of LLMs as reasoning agents with deterministic CPG query engines and GNN structural encoders represents a particularly promising direction. Furthermore, explanation faithfulness, counterfactual reasoning, and adversarial robustness must transition from post-hoc evaluation criteria to first-class training objectives. Finally, closed-loop vulnerability remediation where detection, localization, and verified patch generation form a unified graph-conditioned pipeline represents the next evolutionary step toward autonomous software security. In conclusion, graph neural networks for interprocedural vulnerability detection stand at a pivotal juncture. The theoretical foundations are mature, the architectural toolkit is rich, and the empirical evidence demonstrates substantial progress. Yet the gap between benchmark performance and real-world deployment readiness remains significant. Closing this gap requires coordinated advances in graph construction efficiency, long-range dependency modeling, dataset curation practices, interpretability guarantees, and adversarial resilience. By synthesizing these research threads, this review aims to provide a structured roadmap for building the next generation of scalable, trustworthy, and actionable vulnerability detection systems.

References

Apuri, H., Chinthala, M. M. R., Goel, S., & Yepuri, C. (2026, July). InfraAgent: A multi-agent AI framework for predictive deployment failure detection and autonomous remediation in multi-cloud environments. In *2026 18th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)* (pp. 1-6). IEEE.

Baldwin, C., MacCormack, A., & Rusnak, J. (2014). Hidden structure: Using network methods to map system architecture. *Research Policy*, *43*(8), 1381-1397.

Bhatia, R., & Arora, V. (2012). Evaluation of flow graph and dependence graphs for program representation. *International Journal of Computer Applications*, *56*(14), 18-23.

Black, P. E. (2018). A software assurance reference dataset: Thousands of programs with known bugs. *Journal of research of the National Institute of Standards and Technology*, *123*, 1.

Borowski, K., Balis, B., & Orzechowski, T. (2024). Semantic code graph an information model to facilitate software comprehension. *IEEE Access*, *12*, 27279-27310.

Brown, J., Kahn, M., & Toh, S. (2013). Data quality assessment for comparative effectiveness research in distributed data networks. *Medical care*, *51*(8 0 3), S22.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Wu, X., ... & Liu, W. (2023). Learning to detect memory-related vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, *33*(2), 1-35.

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 Transactions on Software Engineering*, *50*(8), 2163-2177.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J. G., & Chen, W. (2022). Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Chen, Y., Bian, Y., Han, B., & Cheng, J. (2024). How interpretable are interpretable graph neural networks?. *arXiv preprint arXiv:2406.07955*.

Chen, Z., Chu, Z., Gui, Y., Guo, F., Wan, Y., & Shi, C. (2026, July). CGBridge: Bridging Code Graphs and Large Language Models for Better Structure-Aware Code Understanding. In *Findings of the Association for Computational Linguistics: ACL 2026* (pp. 8945-8966).

Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(3), 1-33.

Chumo, J. K. (2026). Evaluating the Robustness of GNN-Based Vulnerability Detectors Under Semantics-Preserving Code Obfuscation.

Dai, P., Zhu, H., Wu, J., & He, H. (2025). An integrated graph neural network model for joint software defect prediction and code quality assessment. *Scientific Reports*, *16*(1), 1677.

Dandibhotla, S., Samudrala, M., Kaneriya, A., & Dakshanamurthy, S. (2025). Gnnseq: A sequence-based graph neural network for predicting protein–ligand binding affinity. *Pharmaceuticals*, *18*(3), 329.

Devkota, S., & Isaacs, K. E. (2018, June). CFGExplorer: Designing a visual control flow analytics system around basic program analysis operations. In *Computer Graphics Forum* (Vol. 37, No. 3, pp. 453-464)

Dhawas, P., Dhore, A., Bhagat, D., Pawar, R. D., Kukade, A., & Kalbande, K. (2024). Big data preprocessing, techniques, integration, transformation, normalisation, cleaning, discretization, and binning. In *Big data analytics techniques for market intelligence* (pp. 159-182). IGI Global Scientific Publishing.

Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., ... & Chen, Y. (2024). Vulnerability detection with code language models: How far are we?. *arXiv preprint arXiv:2403.18624*.

Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., & Wang, G. (2019). Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)* (pp. 869-885).

Du Bois, K., Sartor, J. B., Eyerman, S., & Eeckhout, L. (2013). Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. *ACM SIGPLAN Notices*, *48*(10), 355-372.

Fan, W., & Tian, C. (2022). Incremental graph computations: Doable and undoable. *ACM Transactions on Database Systems (TODS)*, *47*(2), 1-44.

Fu, M., & Tantithamthavorn, C. (2022, May). Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th international conference on mining software repositories* (pp. 608-620).

Fu, R., Qi, M., Meng, C., Yin, S., Liu, K., Kang, Z., & Fong, S. (2026). AdvSynGNN: structure-adaptive graph neural nets via adversarial synthesis and self-corrective propagation. *arXiv preprint arXiv:2602.17071*.

Gavette, J. V., Stoop, M., Hud, N. V., & Krishnamurthy, R. (2016). RNA–DNA chimeras in the context of an RNA world transition to an RNA/DNA world. *Angewandte Chemie*, *128*(42), 13398-13403.

George, G., & Thampi, S. M. (2018). A graph-based security framework for securing industrial IoT networks from vulnerability exploitations. *IEEE access*, *6*, 43586-43601.

Gong, Y., Liu, G., Xue, Y., Li, R., & Meng, L. (2023). A survey on dataset quality in machine learning. *Information and Software Technology*, *162*, 107268.

Grove, D., & Chambers, C. (2001). A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *23*(6), 685-746.

Grove, D., DeFouw, G., Dean, J., & Chambers, C. (1997, October). Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 108-124).

He, H., Xu, H., Zhao, G., Huang, G., & Zhao, B. (2025). Towards structured semantic representation via hierarchical transformer network for software defect prediction. *Journal of Systems and Software*, 112672.

He, Y., Haikun, L., Li, X., Lulu, Z., Zhang, L., Dengqi, Y., & Li, X. (2026). Graph Neural Networks for Software Vulnerability Mining: A Review. *Information*, *17*(8), 730.

Hin, D., Kan, A., Chen, H., & Babar, M. A. (2022, May). Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories* (pp. 596-607).

Horwitz, S., Reps, T., & Binkley, D. (2004). Interprocedural slicing using dependence graphs. *Acm Sigplan Notices*, *39*(4), 229-243.

Hu, Y., Wang, S., Li, W., Peng, J., Wu, Y., Zou, D., & Jin, H. (2023, July). Interpreters for GNN-based vulnerability detection: Are we there yet?. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis* (pp. 1407-1419).

Kaplan, R. M., Whitelock, P., Wood, M. M., Somers, H. L., Johnson, R., & Bennett, P. (1995). Three seductions of computational psycholinguistics. *Formal issues in lexical-functional grammar*, *47*.

Khamaiseh, S. Y., Bagagem, D., Al-Alaj, A., Mancino, M., & Alomari, H. W. (2022). Adversarial deep learning: A survey on adversarial attacks and defense mechanisms on image classification. *IEEE Access*, *10*, 102266-102291.

Kosan, M., Verma, S., Armgaan, B., Pahwa, K., Singh, A. K., Medya, S., & Ranu, S. (2024, May). Gnnx-bench: Unravelling the utility of perturbation-based gnn explainers through in-depth benchmarking. In *International Conference on Learning Representations* (Vol. 2024, pp. 52931-52960).

Lee, K., Ippolito, D., Nystrom, A., Zhang, C., Eck, D., Callison-Burch, C., & Carlini, N. (2022, May). Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 8424-8445).

Lekssays, A. (2026, April). Bridging Code Property Graphs and Language Models for Program Analysis. In *Proceedings of the 2026 IEEE/ACM 4th International Workshop on Software Vulnerability Management* (pp. 33-40).

Li, J., Zhao, B., & Zhang, C. (2018). Fuzzing: a survey. *Cybersecurity*, *1*(1), 6.

Li, K., Wang, R., Zhang, T., & Ishibuchi, H. (2018). Evolutionary many-objective optimization: A comparative study of the state-of-the-art. *Ieee Access*, *6*, 26194-26214.

Liu, M., Wang, Z., Chen, Z., Pei, Z., Wang, Y., & Zheng, Z. (2026). Dynamic analysis enhances issue resolution. *arXiv preprint arXiv:2603.22048*.

Liu, S. (2020, December). A unified framework to learn program semantics with graph neural networks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1364-1366).

Lukovnikov, D., & Fischer, A. (2021, July). Improving breadth-wise backpropagation in graph neural networks helps learning long-range dependencies. In *International Conference on Machine Learning* (pp. 7180-7191). PMLR.

McDermott, M. B., Steinberg, E., Fries, J. A., van de Water, R. P., Pang, C., Rockenschaub, P., ... & Shah, N. H. (2026). MEDS An Emerging Data Standard and Ecosystem for Health AI Research. *NEJM AI*, *3*(6), AIra2501253.

Mienye, I. D., & Swart, T. G. (2024). A comprehensive review of deep learning: Architectures, recent advances, and applications. *Information*, *15*(12), 755.

Naeem, M. R., Lin, T., Naeem, H., Ullah, F., & Saeed, S. (2019). Scalable mutation testing using predictive analysis of deep learning model. *IEEE Access*, *7*, 158264-158283..

Ni, C., Shen, L., Yang, X., Zhu, Y., & Wang, S. (2024, April). MegaVul: AC/C++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 21st International Conference on mining software repositories* (pp. 738-742).

Papenmeier, A., Englebienne, G., & Seifert, C. (2019). How model accuracy and explanation fidelity influence user trust. *arXiv preprint arXiv:1907.12652*.

Paul, D., West, R., Bosselut, A., & Faltings, B. (2024, November). Making reasoning matter: Measuring and improving faithfulness of chain-of-thought reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2024* (pp. 15012-15032).

Perez-Botero, D., Szefer, J., & Lee, R. B. (2013, May). Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing* (pp. 3-10).

Qasem, A., Debbabi, M., & Soeanu, A. (2024, December). Octopustaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (pp. 2355-2369).

Rahman, R., & Nguyen, D. C. (2026). Quantum convolutional neural networks: A survey on architectures, applications, and future directions. *IEEE Transactions on Neural Networks and Learning Systems*.

Rao, X., Sun, C., Zeng, D., Huang, Y., & Tan, G. (2024). Sliver: a scalable slicing-based verification for information flow security. *IEEE Transactions on Dependable and Secure Computing*, *22*(1), 457-473.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016, August). " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 1135-1144).

Roesler, M., Wells, C., Schamberg, G., Gao, J., Boyle, L., Harrison, E., ... & Varghese, C. (2026). Class imbalance correction in artificial intelligence models leads to miscalibrated clinical predictions: a real-world evaluation. *medRxiv*, 2026-03.

Romero, O. J., Zimmerman, J., Steinfeld, A., & Tomasic, A. (2023). Synergistic integration of large language models and cognitive architectures for robust ai: An exploratory analysis. In *Proceedings of the AAAI Symposium Series* (Vol. 2, No. 1, pp. 396-405).

Seo, M., Choi, W., You, M., & Shin, S. (2025). Autopatch: Multi-agent framework for patching real-world cve vulnerabilities. *arXiv preprint arXiv:2505.04195*.

She, X., Liu, Y., Zhao, Y., He, Y., Li, L., Tantithamthavorn, C., ... & Wang, H. (2026). Pitfalls in language models for code intelligence: A taxonomy and survey. *ACM Transactions on Software Engineering and Methodology*, *35*(3), 1-42.

Shuvo, M. I. H. (2026). Detecting Interprocedural Software Vulnerabilities Using Causal Deep Learning.

Shuvo, M. I. H. (2026). Detecting Interprocedural Software Vulnerabilities Using Causal Deep Learning.

Shuvo, M. I. H. (2026). Detecting Interprocedural Software Vulnerabilities Using Causal Deep Learning.

Shuvo, M. I. H. (2026). Detecting Interprocedural Software Vulnerabilities Using Causal Deep Learning.

Siewruk, G., & Berej, M. (2026). Transformation of Vulnerability Management through Artificial Intelligence: An Overview of Generative and Learning Models. *IEEE Access*.

Soni, C. B. (2026). *Cluster-Aware Vulnerability Detection: A Systematic Evaluation Across Imbalance Ratios* (Master's thesis, Wright State University).

Sulc, A. (2025). Event2Vec: A Geometric Approach to Learning Composable Representations of Event Sequences. *arXiv preprint arXiv:2509.12188*.

Tiftikci, M., Sharifloo, A. M., & Mezini, M. (2026, April). Deep Graph-Language Fusion for Structure-Aware Code Generation. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering* (pp. 67-71).

Tip, F. (1994). *A survey of program slicing techniques* (p. 58). Amsterdam: Centrum voor Wiskunde en Informatica.

Wang, J., Wu, Y., Zhou, G., Yu, Y., Guo, Z., & Xiong, Y. (2020, November). Scaling static taint analysis to industrial soa applications: A case study at alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1477-1486).

Wang, X., Hu, R., Gao, C., Wen, X. C., Chen, Y., & Liao, Q. (2024, April). Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (pp. 472-483).

Wei, J., Zhang, Y., Chen, Y., Ma, P., Liang, L., Zhang, C., & Zhou, J. (2026). MDFVD: vulnerability detection via multi-dimensional-feature-based code representation. *Journal of King Saud University Computer and Information Sciences*.

Wu, B., Li, J., Yu, J., Bian, Y., Zhang, H., Chen, C., ... & Zhao, P. (2022). A survey of trustworthy graph learning: Reliability, explainability, and privacy protection. *arXiv preprint arXiv:2205.10014*.

Xu, Y., & Zhu, Y. (2022). A survey on pretrained language models for neural code intelligence. *arXiv preprint arXiv:2212.10079*.

Yadavally, A., Li, Y., Wang, S., & Nguyen, T. N. (2024). A learning-based approach to static program slicing. *Proceedings of the ACM on Programming Languages*, *8*(OOPSLA1), 83-109.

Yang, M., Zhou, M., Zhang, T., Liu, J., Li, Z., Pan, L., ... & King, I. (2022). Hyperbolic graph neural networks: A review of methods and applications. *arXiv preprint arXiv:2202.13852*.

Yang, Y., Xu, L., Zhou, M., & Li, L. (2025, November). Context-Sensitive Pointer Analysis for ArkTS. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 3262-3273). IEEE.

Yun, I., Lee, S., Xu, M., Jang, Y., & Kim, T. (2018). {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)* (pp. 745-761).

Zhang, Z., & Barthe, G. (2025). CT-LLVM: Automatic large-scale constant-time analysis. *Cryptology ePrint Archive*.

Zhou, M., Zhang, Y., Han, J., Li, L., & Luo, X. (2026). Security of Cross-Platform and Cross-Language Software Technologies: A Systematic Review. *ACM Transactions on Software Engineering and Methodology*.

Zhou, W. (2026, April). The Natural Geometry of Code: Hyperbolic Representation Learning for Program Reasoning. In *International Conference on Learning Representations* (Vol. 2026, pp. 38220-38264).

Zhou, Z., Ji, Y., Xu, Z., Zheng, Z., Zhou, S., Wen, W., ... & Wang, C. (2026, August). MOBI: Monolithic Graph-Language Modeling Beyond Modality Interference. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2* (pp. 6911-6922).



2026 by the authors; Journal of *Kashmir Journal of Academic Research and Development*. This is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) license (http://creativecommons.org/licenses/by/4.0/).