

Review

Graph Neural Networks for Software Vulnerability Mining: A Review

Yuan He, Haikun Lv , Xing Li, Lulu Zeng, Lina Zhang *, Dengqi Yang and Xiaowei Li

College of Mathematics and Computer Science, Dali University, Dali 671003, China; hey@dali.edu.cn (Y.H.); lvhk000704@163.com (H.L.); 13078721020@163.com (X.L.); 15008712224@163.com (L.Z.); dengqiyang@163.com (D.Y.); lixiaowei_xidian@163.com (X.L.)

* Correspondence: zln@dali.edu.cn

Abstract

Software vulnerability mining is important for improving software reliability and security. Compared with sequence-based models, graph neural networks (GNNs) can explicitly model program structures such as syntax, control flow, data flow, call relations, and dependency paths, and have therefore been widely studied for vulnerability detection, localization, explanation, and repair. This paper presents a PRISMA-informed structured topical review of 87 studies and addresses five research questions concerning program graph representation, homogeneous and heterogeneous GNN architectures, Graph-LLM integration, evaluation reliability, and future research directions. The reviewed evidence shows that graph-based methods are most effective when vulnerability mechanisms can be faithfully represented through explicit structural relations. However, their reported performance remains strongly affected by duplicated samples, random function-level splits, noisy labels, incomplete repository context, graph-construction errors, and weak explanation protocols. Homogeneous GNNs provide efficient structural message passing but may mix different semantic relations, whereas heterogeneous GNNs preserve relation types more explicitly at the cost of greater graph-quality and computational requirements. Graph-LLM systems can improve semantic reasoning, repository-level analysis, explanation generation, and repair support, but their benefits should be evaluated together with memory consumption, inference latency, deployment complexity, and verification cost. This review further proposes minimum requirements for reliable vulnerability benchmarks and verifiable explanations, and develops a strategic agenda covering leakage-resistant datasets, uncertainty-aware graph construction, repository-level evaluation, cost-effective Graph-LLM collaboration, and graph-guided autonomous vulnerability repair.

Keywords: graph neural networks; software vulnerability mining; program graph representation; code property graph; heterogeneous graph learning; Graph-LLM integration; evaluation protocols



Academic Editor: Kyriakos Kritikos

Received: 23 June 2026

Revised: 22 July 2026

Accepted: 24 July 2026

Published: 28 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

With the widespread adoption of open-source reuse, component-based development, and continuous integration, modern software systems increasingly depend on third-party libraries and external components. This trend improves development efficiency but also expands the attack surface of software systems. Vulnerabilities may exist not only in application code but also in dependent components, build scripts, and software supply chains, making vulnerability mining an important task for software security assurance [1–3].

Traditional vulnerability mining methods mainly rely on rule matching, program slicing, symbolic execution, fuzzing, and static analysis. These methods have clear analytical procedures and are still widely used in practice. However, they often depend on expert-defined rules, precise program analysis, or carefully designed test inputs. In large and complex software systems, such methods may suffer from high false-positive rates, incomplete path coverage, imprecise call graph construction, and limited scalability [4–11]. Therefore, learning-based vulnerability mining has attracted increasing attention in recent years.

Programs contain rich structural information. Abstract syntax trees, control-flow graphs, data-flow graphs, program-dependence graphs, and code property graphs can describe source code from different perspectives, including syntactic organization, execution paths, data dependencies, and semantic relations [12–15]. Compared with purely sequence-based representations, graph-based representations are better suited to modeling cross-statement dependencies and structural propagation patterns. Based on these representations, graph neural networks (GNNs) have been widely applied to vulnerability detection, localization, explanation, smart contract analysis, and graph-enhanced code reasoning [16–19].

Although GNN-based methods have achieved promising results on public benchmarks, quantitative evidence indicates that commonly used evaluation settings may substantially overestimate practical vulnerability-detection capability. Chakraborty et al. reported that existing datasets contained up to 68% duplicated samples and that model performance dropped by more than 50% under a more realistic evaluation setting [20]. A subsequent survey of 81 recent ML-based vulnerability-detection studies found that 88% formulated the task as function-level binary classification [21]. In a manual examination of 100 functions labeled as vulnerable from each of BigVul, Devign, and DiverseVul, only 39%, 47%, and 65%, respectively, were confirmed to contain the reported vulnerabilities. Moreover, all 151 confirmed vulnerable functions required information outside the isolated function for accurate identification. A classifier using only word-count features still achieved F1 scores of 0.86 on BigVul and 0.622 on Devign, indicating that high benchmark scores can be obtained from spurious dataset correlations without modeling program structure or vulnerability semantics [21].

These findings motivate stricter benchmark construction and repository-level evaluation. ReposVul, for example, contains 6,134 CVE entries covering 236 CWE types and 1491 projects, and preserves repository-, file-, function-, and line-level information to support analysis beyond isolated functions [22]. Therefore, it is necessary to review not only existing GNN architectures but also the dataset assumptions, graph-construction fidelity, evaluation protocols, and contextual conditions under which their reported advantages remain valid.

This paper reviews graph neural networks for software vulnerability mining, covering source-code vulnerability detection, vulnerability localization, vulnerability explanation, smart contract analysis, and graph-enhanced code reasoning. Java and smart contracts are selected as two representative analytical cases rather than as the exclusive scope of the review.

Java represents general-purpose, object-oriented software systems in which vulnerability analysis is complicated by inheritance, dynamic dispatch, reflection, serialization, framework callbacks, and inter-procedural dependencies. These language and framework mechanisms can introduce missing call relations, imprecise dependency edges, and incomplete program contexts during static graph construction [23–27]. Java therefore provides a representative setting for examining graph-construction fidelity and the capability boundaries of GNN-based vulnerability mining.

Smart contracts represent security-critical programs whose vulnerabilities are often associated with explicit control-flow, data-flow, call, transaction-state, and inter-contract relations. Reentrancy, access-control errors, unsafe external calls, and state-dependent execution paths make graph-based structural reasoning particularly relevant in this setting [28–32]. Smart contracts therefore provide a complementary case for analyzing heterogeneous relations, graph-guided reasoning, and security-specific structural constraints.

Other programming languages, including C/C++, Python and JavaScript are also included when the corresponding studies contribute to the research questions of this review. Java and smart contracts are used as focused representative cases to support deeper analysis of two distinct scenarios: complex general-purpose software systems and blockchain-oriented security-critical programs. Binary-only vulnerability analysis, penetration testing, malware detection, and unrelated graph-learning applications are outside the scope unless they directly contribute to software vulnerability mining.

Several high-quality reviews have examined research directions closely related to software vulnerability mining. Ghaffarian and Shahriari surveyed machine-learning-based and data-mining-based techniques for software vulnerability analysis and discovery [33]. Lin et al. subsequently reviewed deep neural network-based vulnerability detection methods, with particular attention to code representations, neural architectures, datasets, and evaluation settings [34]. More recently, Shiri Harzevili et al. conducted a systematic literature review of automated software vulnerability detection using machine learning and organized the field according to data sources, input representations, learning models, and detection tasks [35]. Chu et al. reviewed smart contract vulnerabilities from the perspectives of vulnerability data sources, detection techniques, and repair methods [36]. Sheng et al. further examined the application of large language models to software vulnerability detection, including model selection, application strategies, datasets, and evaluation metrics [37].

These reviews provide valuable taxonomies of machine-learning models, deep neural networks, source-code representations, smart contract security techniques, and LLM-based vulnerability detection. However, their analytical scopes are primarily organized around individual model families or application domains. They provide limited joint analysis of how dataset construction, program graph fidelity, homogeneous and heterogeneous graph architectures, Graph-LLM integration, evaluation protocols, explanation verifiability, and deployment costs interact throughout the complete vulnerability-mining pipeline. In particular, the propagation of errors from duplicated or weakly labeled datasets and incomplete program graphs to unreliable predictions and unverifiable analyst-facing evidence has not been systematically synthesized across homogeneous GNNs, heterogeneous GNNs, and Graph-LLM systems.

To address this gap, this review adopts a pipeline-oriented analytical framework that connects dataset construction, program graph representation, model learning, evaluation reliability, and analyst-facing evidence. Rather than ranking models only according to benchmark accuracy, the review examines the conditions under which graph-based methods provide reliable structural advantages, the circumstances in which their reported gains may be affected by leakage or graph-construction errors, and the evidence required for practical vulnerability localization, explanation, and repair.

The contributions of this review are summarized as follows.

First, it provides a research-question-guided synthesis of program graph representations and node or edge features used in software vulnerability mining, including ASTs, CFGs, DFGs, PDGs, CPGs, multi-view graphs, and heterogeneous program graphs.

Second, it comparatively analyzes homogeneous GNNs, heterogeneous GNNs, semantic-enhanced graph learning, and Graph-LLM integration from the perspectives of structural expressiveness, capability boundaries, computational cost, task granularity, and applicable security scenarios.

Third, it introduces a pipeline-oriented reliability analysis that connects dataset quality, graph-construction fidelity, model design, evaluation protocol, generalization setting, explanation validity, and deployment evidence. This analysis highlights how early-stage data or graph errors may propagate to downstream detection and explanation results.

Fourth, it consolidates minimum requirements for reliable vulnerability benchmarks and verifiable explanations, and develops a strategic research agenda covering leakage-resistant datasets, high-fidelity program graphs, repository-level evaluation, cost-effective Graph-LLM integration, and graph-guided autonomous vulnerability repair.

The remainder of this paper is organized as follows. Section 2 presents the review methodology, including the research questions, search strategy, eligibility criteria, study-selection process, data extraction, quality assessment, and synthesis strategy. Section 3 reviews program graph modeling and node representation methods. Sections 4 and 5 discuss homogeneous and heterogeneous GNN-based vulnerability mining methods, respectively. Section 6 analyzes Graph-LLM integration for vulnerability mining. Section 7 provides a cross-paradigm discussion and strategic research agenda, focusing on benchmark reliability, graph-construction fidelity, explanation validity, applicability boundaries, and future research directions. Section 8 concludes the paper.

2. Review Methodology

2.1. Review Protocol and Scope

This review adopts a PRISMA-informed structured topical review design [38–41]. In this manuscript, a structured topical review refers to a review that combines an explicit literature-search and screening protocol with a topic-centered critical synthesis of technically heterogeneous studies. It differs from a conventional narrative review because the information sources, search terms, eligibility criteria, screening stages, coding dimensions, and quality considerations are explicitly documented. It also differs from a scoping review, whose primary purpose is to map the breadth and characteristics of an evidence field, and from a quantitative meta-analysis, which requires sufficiently comparable empirical outcomes for statistical pooling.

This review design was selected because the examined studies differ substantially in programming languages, vulnerability types, program graph representations, model architectures, datasets, evaluation metrics, task granularities, and experimental protocols. Direct statistical aggregation of the reported results would therefore be methodologically inappropriate. Instead, PRISMA 2020 is used to improve the transparency of study identification, screening, eligibility assessment, and inclusion, while the selected studies are synthesized through research-question-guided thematic analysis and a pipeline-oriented technical classification.

The scope of this review focuses on graph-based and GNN-based methods for software vulnerability mining. Specifically, it covers program graph representations, homogeneous and heterogeneous graph neural networks, semantic-enhanced graph learning, Graph-LLM integration, vulnerability detection, vulnerability localization, explanation-oriented evaluation, benchmark reliability, and practical deployment. Studies unrelated to software vulnerability mining, or studies that do not involve program graph representation, graph-based learning, code-security analysis, or evaluation reliability, are outside the scope of this review.

2.2. Research Questions

To ensure that the literature synthesis is guided by explicit analytical objectives rather than by a descriptive enumeration of prior studies, this review addresses the following research questions:

- RQ1.** What program graph representations and node or edge features have been used for software vulnerability mining, and what vulnerability-relevant information do they preserve?
- RQ2.** How do homogeneous and heterogeneous graph neural network architectures model vulnerability-related structural information, and what are their respective capability boundaries?
- RQ3.** How are graph neural networks integrated with pretrained code models and large language models for vulnerability detection, localization, explanation, reasoning, and repair?
- RQ4.** Which dataset, graph-construction, evaluation, interpretability, and generalization issues affect the reliability and engineering applicability of reported results?
- RQ5.** What research directions are required to support reliable, verifiable, and deployable graph-based software vulnerability mining?

These research questions guide both the data extraction process and the organization of the subsequent synthesis. RQ1 is addressed in Section 3, RQ2 in Sections 4 and 5, RQ3 in Section 6, and RQ4 and RQ5 in Section 7. This mapping ensures that the reviewed evidence is interpreted according to the stated analytical objectives.

2.3. Information Sources and Search-Term Development

The literature coverage spans publications from 2014 to July 2026. During the revision stage, the literature set was updated to incorporate newly available studies relevant to the defined research questions. The year 2014 was selected as the starting point because research on code property graphs provided an important foundation for graph-based software vulnerability analysis. The information sources included IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink, and arXiv. Backward and forward snowballing was also conducted to identify relevant studies that were not retrieved through database searches alone.

The search terms were developed through a four-step process. First, representative seed studies on program graphs, graph neural networks, and software vulnerability detection were examined to identify commonly used terminology. Second, the identified terms were organized into four concept groups: graph learning, program representation, vulnerability analysis, and semantic-enhanced modeling. Third, pilot searches were conducted in the selected databases, and the search expressions were iteratively refined by adding synonyms, abbreviations, model names, and task-specific terms found in relevant records. Fourth, backward and forward snowballing was applied to the included seed studies and subsequently selected papers to identify additional terminology and relevant publications. This process was used to reduce dependence on subjective keyword selection and improve search coverage.

The graph-learning concept group included terms such as “graph neural network”, GNN, GCN, GGNN, GAT, HGT, “heterogeneous graph”, and “relational graph neural network”. The program-representation group included “program graph”, “code property graph”, AST, CFG, DFG, PDG, call graph, program slice, and heterogeneous program graph. The vulnerability-analysis group included “vulnerability detection”, “vulnerability mining”, “software vulnerability”, “vulnerability localization”, “vulnerability explanation”, and “smart contract vulnerability”. To cover semantic-enhanced and emerging approaches,

additional terms included “pretrained code model”, CodeBERT, GraphCodeBERT, CodeT5, “large language model”, LLM, Graph-LLM, graph retrieval, and neuro-symbolic reasoning.

The main search expression was constructed by combining at least one term from the graph-learning group, one term from the vulnerability-analysis group, and, where applicable, one term from the program-representation group. A representative search expression was:

(“graph neural network” OR GNN OR GCN OR GGNN OR GAT OR HGT OR “heterogeneous graph”) AND (“vulnerability detection” OR “vulnerability mining” OR “software vulnerability” OR “vulnerability localization” OR “smart contract vulnerability”) AND (“program graph” OR “code property graph” OR AST OR CFG OR DFG OR PDG).

For Graph-LLM and pretrained-code-model studies, supplementary searches combined semantic-model terms with vulnerability-analysis terms, for example:

(“large language model” OR LLM OR “pretrained code model” OR CodeBERT OR GraphCodeBERT OR CodeT5) AND (“vulnerability detection” OR “software security” OR “program analysis”).

All retrieved records were subsequently screened according to their relevance to graph-based software vulnerability mining, program graph construction, graph neural network modeling, evaluation reliability, and explanation-oriented vulnerability analysis.

2.4. Eligibility Criteria and Evidence Types

Studies were included if they satisfied at least one of the following criteria:

1. They proposed, evaluated, or systematically analyzed GNN-based methods for software vulnerability detection, vulnerability localization, vulnerability classification, or vulnerability explanation.
2. They investigated program graph representations, including ASTs, CFGs, DFGs, PDGs, CPGs, multi-view graphs, heterogeneous graphs, call graphs, or program slices, for software vulnerability mining.
3. They studied graph-based vulnerability analysis in smart contracts, repository-level software systems, or other security-critical programming environments.
4. They combined program graphs or graph neural networks with pretrained code models, large language models, retrieval mechanisms, or neuro-symbolic reasoning for vulnerability analysis.
5. They examined evaluation reliability, dataset quality, graph-construction fidelity, generalization, explanation validity, or engineering deployment in learning-based vulnerability mining.

Studies were excluded if they were unrelated to software vulnerability mining, did not involve program graph representation or graph-based learning, lacked sufficient methodological or experimental details, or focused only on unrelated graph-learning applications such as communication networks, recommendation systems, localization, signal processing, or cyber-physical systems. Duplicate records, short abstracts without substantive technical content, inaccessible full texts, and papers whose contribution could not be clearly connected to software vulnerability mining were also excluded.

Peer-reviewed journal and conference papers were treated as the primary evidence base. arXiv preprints and technical reports were included only when they addressed rapidly emerging topics, such as Graph-LLM integration, repository-level vulnerability reasoning, neuro-symbolic analysis, or agent-based vulnerability repair, and when no equivalent peer-reviewed version was available. These non-peer-reviewed studies were clearly distinguished from peer-reviewed publications and were used mainly to describe

recent technical trends and open research directions rather than to support strong empirical conclusions.

When multiple versions of the same study were identified, the peer-reviewed version was retained whenever available. If no peer-reviewed version existed, the most complete and technically detailed version was selected. This evidence-selection policy was adopted to balance methodological reliability with the need to represent rapidly developing research directions.

2.5. Screening and Study Selection

The study-selection process is reported according to the main PRISMA 2020 stages of identification, screening, eligibility assessment, and inclusion. The initial database searches and supplementary backward and forward snowballing identified more than 120 potentially relevant records.

Duplicate publications, clearly irrelevant records, and studies outside the scope of software vulnerability mining were first removed. The remaining records were screened by title and abstract to determine whether they addressed program graph representation, graph-based vulnerability analysis, GNN-based vulnerability detection, Graph-LLM integration, evaluation reliability, or explanation-oriented vulnerability analysis.

The full texts of the potentially eligible studies were then assessed according to the criteria described in Section 2.4. Studies were excluded if they lacked sufficient methodological or experimental detail, did not provide a substantive connection to graph-based software vulnerability mining, were unavailable in complete form, or duplicated a more complete or peer-reviewed publication version.

After the screening and eligibility-assessment stages, 87 studies were included in the final review corpus. The 87-study analytical corpus excludes methodological guidelines and prior review articles cited solely to define the review protocol or position the present review relative to existing surveys. These studies form the evidence base for the subsequent analysis of program graph representations, homogeneous and heterogeneous graph neural networks, semantic-enhanced graph learning, Graph-LLM integration, benchmark reliability, explanation validity, and future research directions.

Because exact stage-level counts were not retained, the review reports the approximate initial retrieval scale and final included corpus without retrospectively estimating unverifiable intermediate counts.

2.6. Data Extraction and Coding

A structured coding form was developed and applied to extract information from each included study. The extracted descriptive fields included publication year, publication type, programming language, vulnerability type, research task, dataset source, sample granularity, program graph representation, graph-construction method, node and edge features, model architecture, baseline methods, evaluation metrics, and reported performance.

The analytical coding fields focused on factors that affect the reliability and applicability of graph-based vulnerability mining. These fields included dataset deduplication, data-split strategy, temporal leakage, project-level or repository-level evaluation, label quality, graph-construction fidelity, inter-procedural context, dynamic language features, cross-project generalization, explanation mechanism, evidence-verification strategy, computational cost, deployment setting, and reported threats to validity.

The coding framework was aligned with the research questions defined in Section 2.2. Program graph representations and node or edge features were coded for RQ1. Homogeneous and heterogeneous GNN architectures, message-passing mechanisms, and capability boundaries were coded for RQ2. Graph-LLM integration, pretrained semantic representa-

tions, graph-guided reasoning, and repair-oriented methods were coded for RQ3. Dataset quality, evaluation protocol, graph fidelity, interpretability, and generalization were coded for RQ4. Open challenges, practical limitations, and proposed future directions were coded for RQ5.

The extracted information was reviewed for internal consistency before synthesis. Ambiguous classifications were rechecked against the full text of the corresponding study, and coding disagreements were resolved through discussion among the authors. The finalized coding results were then used to support the thematic and cross-paradigm synthesis presented in Sections 3–7.

2.7. Quality Appraisal and Potential Risk of Bias

The methodological quality and potential risk of bias of the included studies were considered during full-text coding using six dimensions: research-task clarity, dataset and label provenance, graph-construction transparency, evaluation-protocol reliability, baseline and reproducibility adequacy, and explanation or deployment evidence. Table 1 summarizes the appraisal criteria.

Table 1. Quality-assessment and risk-of-bias criteria for the included studies.

Assessment Dimension	Assessment Criteria
Research-task clarity	Whether the vulnerability definition, research objective, task granularity, programming language, and application scenario were clearly specified.
Dataset and label provenance	Whether the dataset source, labeling process, vulnerability categories, class distribution, duplicate handling, and potential label noise were sufficiently documented.
Graph-construction transparency	Whether the program graph type, construction tool, node and edge definitions, preprocessing steps, and known limitations of static or dynamic analysis were clearly reported.
Evaluation-protocol reliability	Whether the study described its train–validation–test split, deduplication strategy, project-level or temporal separation, cross-project evaluation, and leakage-control measures.
Baseline and reproducibility adequacy	Whether appropriate baseline methods, evaluation metrics, implementation details, hyperparameters, source code, or data-access information were provided.
Explanation and deployment evidence	Whether model explanations were validated using vulnerable lines, program slices, dependency paths, patches, counterfactual tests, static-analysis reports, expert assessment, or realistic deployment evidence.

The assessment criteria were applied qualitatively to guide the interpretation of the included evidence. Studies with clear task definitions, transparent dataset and label provenance, reproducible graph-construction procedures, leakage-aware evaluation protocols, adequate baseline comparisons, and verifiable explanation evidence were treated as providing stronger evidence. Studies with incomplete reporting, possible data leakage, weak baseline comparisons, unclear graph-construction procedures, or unverifiable explanations were interpreted more cautiously.

This appraisal was used to support evidence weighting rather than to assign formal numerical scores or categorical rankings to individual studies. Studies that satisfied the eligibility criteria but contained methodological limitations were retained when they provided technically relevant insights; however, such studies were not used alone to support strong general conclusions. Studies lacking sufficient methodological detail to determine their relevance or reliability were excluded during full-text screening.

2.8. Synthesis Strategy and Mapping to Research Questions

Because the included studies differ substantially in programming languages, vulnerability types, datasets, graph representations, model architectures, evaluation metrics, and task granularities, their reported results were not suitable for direct statistical pooling. The evidence was therefore synthesized through a research-question-guided thematic analysis and a pipeline-oriented comparison.

For RQ1, the included studies were grouped according to program graph representation, graph-construction process, and node or edge features. The synthesis compared ASTs, CFGs, DFGs, PDGs, CPGs, multi-view graphs, and heterogeneous program graphs in terms of the vulnerability-relevant information they preserve and the limitations introduced during graph construction.

For RQ2, the studies were organized according to homogeneous and heterogeneous GNN paradigms. Homogeneous methods were compared in terms of neighborhood aggregation, gated propagation, attention mechanisms, and capability boundaries. Heterogeneous methods were compared according to metapath-based modeling, relation-specific propagation, type-aware attention, graph-quality requirements, and computational complexity.

For RQ3, Graph-LLM studies were synthesized according to three integration routes: graph-enhanced large language models, LLM-enhanced graph learning, and graph-guided or neuro-symbolic reasoning. The comparison considered semantic and structural complementarity, computational cost, inference latency, verification requirements, and applicability to detection, localization, explanation, and repair.

For RQ4, the evidence was analyzed across dataset quality, graph-construction fidelity, evaluation protocol, cross-project generalization, explanation validity, and deployment evidence. Particular attention was paid to whether reported performance could be affected by duplicated samples, random data splits, temporal leakage, noisy labels, incomplete program context, or unverifiable explanations.

For RQ5, recurring limitations and proposed solutions were consolidated into a strategic research agenda covering leakage-resistant benchmarks, high-fidelity graph construction, verifiable explanations, repository-level analysis, cost-effective Graph-LLM integration, and graph-guided autonomous vulnerability repair.

This synthesis strategy enables the review to compare studies beyond model accuracy and to examine how dataset construction, program representation, learning architecture, evaluation design, and analyst-facing evidence jointly affect the reliability and practical value of graph-based vulnerability mining.

Based on the screened literature, structured coding framework, and research-question-guided synthesis, Figure 1 summarizes the technical evolution identified in this review. The framework reflects the transition from rule-based static analysis and explicit program graph construction to homogeneous GNNs, heterogeneous graph learning, and Graph-LLM integration. It was derived after the study-selection and thematic-coding processes rather than being predefined before the review.

2.9. Review Limitations

This review has several methodological limitations. First, although multiple academic databases and backward and forward snowballing were used, some relevant studies may not have been retrieved because of differences in indexing, terminology, publication venue, or database coverage. The search primarily focused on English-language and publicly accessible publications, which may introduce language and availability bias.

Second, the included studies differ substantially in programming languages, vulnerability categories, graph representations, datasets, task granularities, evaluation metrics, and experimental protocols. This heterogeneity prevented meaningful quantitative meta-analysis and required a qualitative thematic synthesis. Consequently, direct numerical comparisons across studies should be interpreted cautiously.

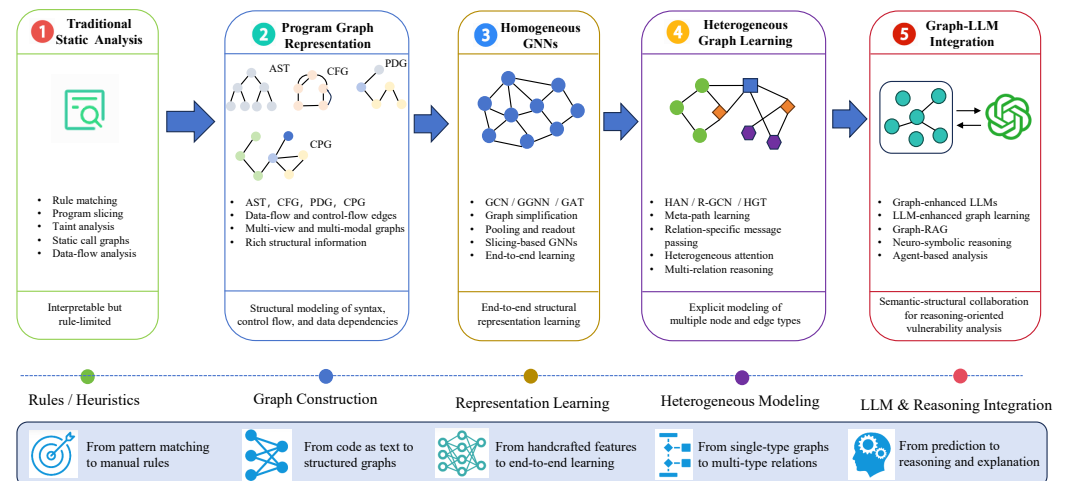


Figure 1. Technical evolution of graph-based vulnerability mining from traditional static analysis and program graph representations to homogeneous GNNs, heterogeneous graph learning, and Graph-LLM integration.

Third, peer-reviewed journal and conference papers were prioritized, but a limited number of preprints and technical reports were retained to represent rapidly emerging topics such as Graph-LLM integration, neuro-symbolic reasoning, and agent-based vulnerability repair. These sources were used mainly to identify recent trends and were not treated as equivalent to peer-reviewed empirical evidence.

Fourth, the quality appraisal relied on the methodological information reported in the original publications. Incomplete reporting may therefore affect the accuracy of the appraisal. Because formal numerical scores or study-level categorical rankings were not assigned, the appraisal was used only to guide cautious interpretation and evidence weighting.

In addition, exact numerical records for every intermediate screening and exclusion stage were not retained because the review protocol was retrospectively formalized as PRISMA-informed. Therefore, the review reports the initial approximate retrieval scale and final included corpus without reconstructing unverifiable intermediate counts.

To reduce these limitations, the review combined multiple information sources, explicit eligibility criteria, snowballing, structured coding dimensions, full-text verification, and cautious interpretation of studies with incomplete reporting or elevated risk of bias. Nevertheless, the conclusions should be understood as a structured synthesis of the available evidence rather than as an exhaustive or statistically pooled estimate of model effectiveness.

3. Program Graph Modeling and Node Representation

Program graph modeling is the basis for applying graph neural networks to software vulnerability mining. Different program graphs describe different aspects of source code, such as syntactic structure, execution paths, data dependencies, and semantic relations. Therefore, the quality of graph construction directly affects the information available to downstream graph learning models. If important dependency edges or relation types are missing, the model may learn incomplete or misleading vulnerability patterns.

In this review, high-fidelity program graph construction refers to the ability of a generated graph to preserve vulnerability-relevant syntactic, control-flow, data-flow, call, type, and contextual relations in the original software system. High fidelity does not simply mean constructing a larger graph. Instead, it emphasizes whether the graph can retain the semantic relations involved in vulnerability formation and propagation. Figure 2 illustrates how graph construction errors may affect representation learning, vulnerability detection, and explanation results.

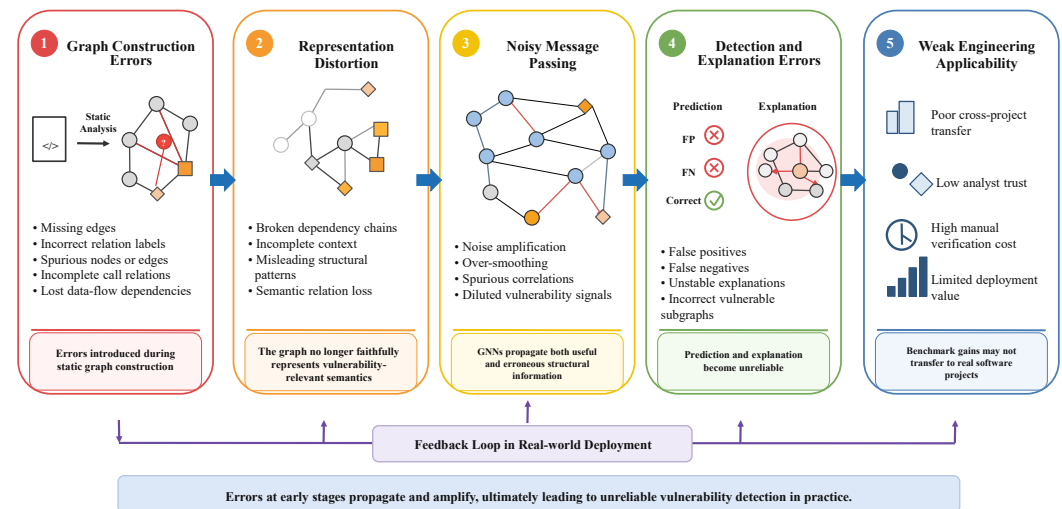


Figure 2. Failure propagation from graph construction errors to vulnerability detection and explanation failures.

3.1. Basic Program Graph Representations

Abstract syntax trees (ASTs), control-flow graphs (CFGs), data-flow graphs (DFGs), and program-dependence graphs (PDGs) are widely used basic program representations. ASTs describe the syntactic hierarchy of source code, CFGs represent possible execution paths, and DFGs or PDGs capture data dependencies and control dependencies [42–45]. These graphs provide structured information for vulnerability analysis, but each representation captures only part of the vulnerability context. ASTs have limited ability to model execution semantics, CFGs do not explicitly describe data dependencies, and PDGs may be costly to construct and incomplete in inter-procedural scenarios.

To integrate multiple program relations, the code property graph (CPG) combines AST, CFG, and PDG information into a unified graph representation [46]. CPG-style representations have been used in vulnerability detection, slicing-based analysis, vulnerable clone detection, smart contract analysis, and graph-enhanced code reasoning [47–52]. By combining syntax, control flow, and data flow, CPGs provide a more comprehensive structural basis for graph-based vulnerability mining. However, their effectiveness still depends on the precision of static analysis. Dynamic dispatch, reflection, framework callbacks, and incomplete dependency information may introduce missing edges or incorrect relations, which can affect downstream vulnerability detection.

Existing program-graph approaches can be grouped according to the type of program information they preserve. Syntax-oriented representations, such as ASTs, describe the hierarchical organization of source code but provide limited execution and dependency semantics. Control-flow-oriented representations, such as CFGs, model possible execution paths and branch transitions. Data- and dependence-oriented representations, including DFGs and PDGs, capture def-use relations, data dependencies, and control dependencies that are often directly related to vulnerability propagation.

Integrated representations, particularly CPGs, combine syntax, control-flow, and data-flow information within a unified graph. Multi-view approaches preserve several graph types separately and fuse their representations during learning, whereas heterogeneous graphs explicitly model different node types, edge types, and semantic relations using type-aware message passing. These approaches differ not only in structural richness but also in graph-construction cost, relation precision, scalability, and vulnerability-context coverage.

Table 2 summarizes the main program-graph approaches used in software vulnerability mining.

Table 2. Common program graph representations in vulnerability mining.

Graph	Captured Information	References
AST	Syntax hierarchy and statement structure	[42,43,53]
CFG	Execution order and control-transfer paths	[42,44,53]
DFG / PDG	Data dependencies, def-use relations, and control dependencies	[42,45,53]
CPG	Syntax, control-flow, data-flow, and dependency relations	[46,50,52,54]
Multi-view graphs	AST, CFG, DFG, PDG, and call graph views	[15,18,53,55]
Heterogeneous graphs	Node types, edge types, and semantic relations	[28–30,56]

As shown in Table 2, different program graph representations capture different aspects of program semantics. ASTs are easy to construct and useful for syntax modeling, but they provide limited information about execution paths and data dependencies. CFGs describe control-transfer relations and are useful for path-sensitive vulnerability patterns, but they do not explicitly represent data flow. DFGs and PDGs further model data dependencies and def-use relations, but their construction may be costly or incomplete in inter-procedural settings. CPGs integrate syntax, control-flow, and data-flow information into a unified representation, but their effectiveness depends heavily on the precision of static analysis. Multi-view and heterogeneous graphs can provide richer structural information, but they also introduce additional challenges such as graph alignment, relation typing, and data sparsity.

3.2. Graph Construction Fidelity in Complex Programs

Graph construction fidelity is especially important in complex language environments. For example, Java programs may involve reflection, dynamic dispatch, serialization, framework callbacks, and JVM-hosted language features. These mechanisms can reduce the completeness and precision of statically constructed program graphs. Studies on Java reflection and static call graph construction show that such mechanisms may lead to missing calls, imprecise call targets, and incomplete dependency relations [23–27].

To improve graph construction quality, existing studies have explored pointer analysis, type-based points-to analysis, serialization-aware call graph construction, taint-based analysis for Java object deserialization, and runtime-information-assisted call graph analysis [57–61]. These techniques can improve specific aspects of graph precision, but they are usually designed under particular assumptions. Therefore, when GNN-based vulnerability mining methods use program graphs as input, the reliability of graph construction should be considered together with model performance.

GNNs can partially mitigate local graph noise through semantic node representations, neighborhood aggregation, attention mechanisms, and learned edge weighting. However, they cannot fully recover vulnerability-relevant program relations that are absent or incorrectly constructed in the input graph. If reflection, dynamic dispatch, framework callbacks, runtime binding, or incomplete inter-procedural analysis causes critical call edges, data-flow paths, or dependency relations to be omitted, downstream message passing is restricted to an incomplete structural view of the program.

Therefore, GNN-based vulnerability mining should not be treated as a substitute for reliable program analysis. Its effectiveness remains bounded by the recall and precision of the underlying graph-construction tools. Future systems should combine static program graphs with runtime traces, points-to or type analysis, uncertainty-aware edge prediction, multi-view graph construction, or dynamic-analysis evidence to reduce the effect of missing and spurious relations.

3.3. Node Representation

In addition to graph structure, node representation also affects graph learning performance. Early graph-based methods often used manually designed features, token embeddings, or syntax-based node attributes. More recent studies use pretrained code models such as CodeBERT, GraphCodeBERT, and CodeT5 to initialize node representations and introduce richer semantic information into graph learning [62–65].

Pretrained node representations can improve the semantic expressiveness of graph models and have been used in vulnerability classification, defect localization, and commit-level analysis [16,62,66,67]. However, semantic initialization does not fully solve the problems of dataset bias, cross-project generalization, or domain shift. Publicly trained code models may not match proprietary codebases, project-specific frameworks, or team-specific coding styles. Therefore, node representation should be analyzed together with graph construction quality and evaluation settings.

Overall, program graph modeling and node representation form the representational foundation of GNN-based vulnerability mining. Basic graphs, CPGs, multi-view graphs, heterogeneous graphs, and pretrained node representations are complementary rather than mutually exclusive. Their effectiveness depends not only on model design, but also on graph fidelity, semantic representation quality, and the evaluation scenario. The following sections further discuss how homogeneous and heterogeneous GNNs use these representations for vulnerability mining.

4. Applications of Homogeneous Graph Neural Networks in Vulnerability Mining

Homogeneous graph neural networks are among the earliest graph learning methods applied to software vulnerability mining. These methods usually transform programs into unified graph structures and learn vulnerability-related representations through message passing. Their main value is to introduce explicit structural information, such as syntax, control flow, data flow, and local dependency context, into neural vulnerability detection. However, this paradigm is also constrained by a unified graph assumption: different program relations are usually aggregated in the same representation space, which weakens the distinction between syntactic adjacency, control dependence, data dependence, and call relations.

In this review, the distinction between homogeneous and heterogeneous graph methods mainly refers to the modeling mechanism. Some GCN-, GGNN-, and GAT-based studies may use CPGs, labeled edges, or multi-graph inputs. However, these relations are usually encoded through unified adjacency matrices, edge attributes, or fused graph views. In contrast, heterogeneous GNNs explicitly assign type-aware or relation-specific parameters to different node types, edge types, or meta-relations.

4.1. GCN, GGNN, and GAT

GCN is a basic route for homogeneous graph-based vulnerability detection. It aggregates neighborhood information on program graphs such as ASTs, CFGs, PDGs, and CPGs, and has been used in vulnerability detection, multi-graph fusion, and graph structure

optimization tasks [18,68–70]. Its advantage is that it provides a simple way to use local graph structure. However, because GCN usually relies on uniform aggregation, it may smooth different semantic relations into similar node representations. This makes it difficult to determine whether a detected pattern is caused by real vulnerability propagation or by superficial structural correlation.

GGNN improves basic graph propagation by introducing gated recurrent message passing. This mechanism is more suitable for modeling dependency paths, program slices, and intermediate representation graphs [55,71,72]. Therefore, GGNN-based methods are often used when vulnerabilities are related to def–use chains, taint propagation, or relatively clear control/data-flow paths. However, this advantage depends heavily on slice quality and graph completeness. If key dependency edges are missing or redundant statements are included, multi-round propagation may amplify graph construction errors. Existing explanation studies also show that the consistency between model explanations and actual vulnerable locations still needs improvement [73,74].

GAT introduces attention mechanisms to assign different weights to neighboring nodes. Compared with uniform aggregation, attention-based aggregation can reduce the influence of some irrelevant neighbors and has been used in cross-project transfer, noise suppression, and semantic enhancement tasks [75–77]. Nevertheless, attention weights mainly reflect statistical relevance under the training data distribution and should not be directly regarded as causal vulnerability evidence. Under the homogeneous graph assumption, GAT still cannot explicitly distinguish control-flow, data-flow, call, and type-related semantic relations.

In addition to these basic models, some studies introduce graph simplification, pooling/readout design, contrastive learning, or pretrained code representations to improve homogeneous GNNs. These designs can alleviate graph noise, weak labels, or insufficient node semantics to some extent. However, they do not fundamentally solve the core limitation of homogeneous message passing: the model still depends on the fidelity of the constructed graph and the reliability of benchmark settings.

4.2. Capability Boundaries

Overall, GCN, GGNN, and GAT reflect the development of homogeneous graph methods from local structural aggregation to dependency-path propagation and selective neighborhood weighting. These methods are useful when vulnerability evidence is mainly contained in local or intra-procedural graph structures. However, real software systems often involve multiple relation types, framework callbacks, dynamic dispatch, inter-procedural dependencies, configuration semantics, and repository-level contexts. In such scenarios, strong results on function-level random-split benchmarks do not necessarily indicate reliable vulnerability mining capability in cross-project or real-world settings.

Therefore, homogeneous GNNs should be viewed as an important but limited stage in graph-based vulnerability mining. Their main contribution is to show that program structure can improve vulnerability representation learning. Their main limitation is that they do not explicitly model multi-type and multi-relational program semantics. This limitation motivates the transition from homogeneous graph learning to heterogeneous graph learning, where node types, edge types, and semantic relations are modeled more explicitly.

Table 3 summarizes representative homogeneous GNN models from the perspectives of modeling mechanism, applicable scenarios, and major limitations.

Table 3. Comparative analysis of homogeneous GNN models in vulnerability mining.

Model	Main Mechanism and Use	Major Limitation
GCN	Aggregates local neighborhood information on unified program graphs such as ASTs, CFGs, PDGs, or CPG-style graphs. It is mainly used to capture local syntactic or structural vulnerability patterns in graph-based detection and graph fusion tasks.	It has limited ability to model long-range dependency propagation. Different semantic relations may be over-smoothed during uniform aggregation [18,68–70].
GGNN	Uses gated recurrent message passing over dependency paths, program slices, or intermediate representation graphs. It is often used for vulnerabilities related to def–use chains, taint propagation, and path-sensitive control/data-flow structures.	It strongly depends on slice quality and graph completeness. Missing edges, redundant nodes, and noisy slices may be amplified through multi-round propagation [55,71–74].
GAT	Introduces attention-based aggregation to assign different weights to neighboring nodes. It is used for selective neighborhood aggregation, partial graph-noise suppression, semantic enhancement, and cross-project transfer.	Attention weights mainly reflect statistical relevance rather than causal vulnerability evidence. Semantic edge types remain implicit under homogeneous graph modeling [75–77].
Enhanced homogeneous GNNs	Combine homogeneous GNNs with graph simplification, pooling/readout design, contrastive learning, or pretrained node representations. These designs are used to alleviate graph noise, weak labels, insufficient node semantics, and limited labeled samples.	They can improve representation learning under specific settings, but cannot fully solve graph fidelity, dataset bias, or cross-project generalization problems.

5. Applications of Heterogeneous Graph Neural Networks in Vulnerability Mining

Heterogeneous graph neural networks extend graph-based vulnerability mining by explicitly modeling different node types, edge types, and semantic relations in program graphs. This direction is mainly motivated by a limitation of homogeneous GNNs: control flow, data flow, call relations, def–use relations, and syntactic dependencies may have different security meanings, but they are often aggregated in a unified representation space. By introducing type-aware or relation-specific modeling, heterogeneous GNNs attempt to preserve these semantic differences during message passing.

However, heterogeneous graph learning should not be regarded as a complete solution to vulnerability mining. Its effectiveness depends on the quality of graph construction, the rationality of relation partitioning, the design of metapaths or type-aware propagation mechanisms, and the availability of sufficient labeled samples. If node or edge types are inaccurate, or if key dependencies are missing from the constructed graph, a more complex heterogeneous model may still learn distorted vulnerability patterns.

5.1. Metapath-Based Heterogeneous Graph Attention Networks

Heterogeneous graph attention networks (HANs) model high-level semantic relations through predefined metapaths and hierarchical attention mechanisms [78]. In vulnerability mining, HAN-based methods have been used for multi-granularity program structure fusion and multi-semantic relation selection. Representative studies such as MAGNET, MANDO, SCVHunter, and HGAN4VD show how metapath-based modeling can introduce explicit semantic paths into program graph learning [28,29,79–81].

The key feature of HAN is that it transforms heterogeneous program semantics into selected metapath instances. This is useful when vulnerability propagation follows relatively clear semantic paths, such as specific combinations of syntax, control-flow, and data-flow relations. However, this design also introduces a strong prior assumption: the manually selected metapaths must be able to cover the actual vulnerability mechanism. If the metapaths are incomplete, too coarse, or inconsistent with the real propagation process, the model may focus on structurally plausible but security-irrelevant paths. Therefore,

HAN-based methods are more suitable for scenarios where program relations are well understood and semantic paths can be reasonably specified.

5.2. Relational Graph Convolutional Networks

Relational graph convolutional networks (R-GCNs) assign relation-specific parameters to different edge types and perform message passing over multi-relational graphs [82]. Compared with HAN, R-GCN reduces the dependence on predefined metapaths and directly uses explicit relation types, such as control dependencies, data dependencies, call relations, and def–use relations. In vulnerability detection, HeVulD and related studies indicate that relation-specific propagation can better preserve vulnerability-relevant structural contexts than fully unified message passing [56,83].

The limitation of R-GCN lies in its dependence on relation labeling quality. If relation types are too coarse, different security meanings may still be mixed; if they are too fine-grained, the model may face parameter growth, data sparsity, and overfitting. In addition, R-GCN mainly handles explicit and enumerable program relations. Vulnerabilities involving implicit business constraints, configuration semantics, framework behaviors, or cross-module conditions are still difficult to represent only through predefined relation types.

5.3. Adaptive Heterogeneous Graph Transformer

Heterogeneous graph transformer (HGT) models interactions among source node type, edge type, and target node type through type-aware attention and heterogeneous message passing [84]. This design allows HGT to learn adaptive importance weights over different heterogeneous relations, rather than relying only on fixed metapaths or independent relation-specific convolutions. In smart contract analysis, MANDO-HGT demonstrates the potential of HGT in rich heterogeneous graph settings [30]. Related heterogeneous graph studies further support the value of typed structural and semantic modeling [31,32].

Compared with HAN and R-GCN, HGT provides more flexible type-aware modeling. Nevertheless, this flexibility also increases the requirements for graph quality, labeled data, and computational resources. Type-aware attention can improve representation capacity, but it does not automatically provide reliable vulnerability explanations. The learned attention weights still need to be validated against real vulnerable statements, dependency paths, or security rules. Therefore, HGT is more suitable for structurally complex vulnerability mining tasks, while its scalability and interpretability remain important issues in realistic software engineering scenarios.

5.4. Summary of Heterogeneous Graph Methods

Overall, heterogeneous graph methods represent a transition from unified structural message passing to explicit multi-relational program semantics modeling. HAN emphasizes metapath-based semantic selection, R-GCN emphasizes relation-specific propagation, and HGT introduces type-aware adaptive message passing. These methods can alleviate the semantic-mixing problem of homogeneous GNNs, but they introduce new dependencies on graph construction fidelity, relation design, data scale, and evaluation protocols.

Therefore, the future value of heterogeneous GNNs does not lie simply in building more complex program graphs or deeper models. A more important direction is to construct high-fidelity program graphs, define security-relevant relation types, combine structural modeling with semantic representations, and evaluate models under stricter cross-project, cross-version, and repository-level settings. Table 4 summarizes representative heterogeneous GNN models used in vulnerability mining.

Table 4. Comparison of typical heterogeneous GNN methods in vulnerability mining.

Method	Mechanism	Main Limitation	References
HAN	Metapath attention	Handcrafted metapaths	[28,29,78–81]
R-GCN	Relation-specific propagation	Relation-label quality	[56,82,83]
HGT	Type-aware attention	Cost and typed-graph quality	[30,84]

Table 4 summarizes three typical heterogeneous GNN methods used in vulnerability mining. HAN uses metapath-based semantic attention to model predefined semantic paths and supports multi-granularity relation selection. However, its effectiveness depends on handcrafted metapaths, and incomplete metapath design may miss real vulnerability propagation paths. R-GCN performs relation-specific message passing and can preserve explicit edge-type information, such as control-flow, data-flow, call, and def-use relations. However, it requires accurate relation labels, and overly fine-grained relations may introduce sparsity and overfitting. HGT further introduces type-aware heterogeneous attention and can model interactions among different node types and edge types without fixed metapaths. However, it requires high-quality typed graphs, sufficient labeled data, and higher computational cost.

6. Graph–LLM Integration for Vulnerability Mining

The integration of graph-based methods and large language models (LLMs) has recently become an emerging direction in software vulnerability mining. The motivation is not that LLMs can simply replace program graphs or GNNs, but that graphs and LLMs provide complementary types of information. Program graphs can provide explicit structural evidence, such as control flow, data flow, call relations, and dependency paths, while LLMs can provide broader semantic representations, contextual understanding, and natural-language reasoning capabilities. Graph–LLM integration is therefore better understood as an attempt to combine structural constraints with semantic reasoning.

Current studies can be broadly divided into three integration routes: graph-enhanced LLMs, LLM-enhanced graph learning, and graph-guided or neuro-symbolic reasoning.

6.1. Graph-Enhanced Large Language Models

In graph-enhanced LLM approaches, program graphs are transformed into prompts, retrieval contexts, serialized graph descriptions, graph embeddings, or auxiliary structural inputs for LLM-based vulnerability analysis [85–87]. This design may help LLMs move beyond plain-text code understanding and focus on vulnerability-relevant structural contexts, including control-flow paths, data dependencies, call relations, and inter-procedural connections.

The principal advantage of this route is that explicit graph evidence can constrain the semantic reasoning process. Instead of analyzing an entire code fragment without structural guidance, the LLM can be provided with selected program slices, dependency paths, suspicious subgraphs, or retrieved graph contexts. This may improve vulnerability localization, explanation generation, and repository-level reasoning.

However, the effectiveness of graph-enhanced LLMs remains dependent on the correctness and completeness of the input graph. If the graph contains missing edges, incorrect dependencies, incomplete call relations, or noisy structural information, the injected graph context may mislead the LLM rather than improve its reasoning. Graph serialization and prompt construction may also discard relation types, node attributes, or long-range dependencies when complex graphs are converted into limited textual contexts. Therefore, graph-enhanced LLM systems should report how graph information is selected, encoded, truncated, and verified.

6.2. LLM-Enhanced Graph Learning

In LLM-enhanced graph learning, the graph model remains responsible for structural propagation, while pretrained code models or LLMs provide semantic node features, path representations, code summaries, or global contextual information [88–90]. This design can enrich the semantic expressiveness of GNN-based vulnerability detectors and partially reduce the limitations of manually designed node features or token-level embeddings.

Pretrained semantic representations are particularly useful when structurally similar code fragments have different security meanings or when vulnerability evidence depends on identifiers, API semantics, surrounding statements, or natural-language descriptions. Semantic features can also support heterogeneous graph learning by providing richer representations for different node types, such as statements, variables, functions, contracts, and external calls.

Nevertheless, semantic enhancement cannot replace high-fidelity graph construction. If node types, edge types, call relations, or dependency paths are incorrectly constructed, pretrained representations cannot fully recover the missing program semantics. The combination of LLMs and GNNs also introduces additional training cost, inference cost, memory consumption, and system complexity. Moreover, performance gains may be affected by pretraining-data leakage, domain mismatch, and inconsistent alignment between token-level semantic representations and graph-level structural elements.

6.3. Graph-Guided and Neuro-Symbolic Reasoning

Graph-guided reasoning and neuro-symbolic approaches use program graphs, static-analysis results, knowledge graphs, retrieval mechanisms, security rules, or dependency paths as reasoning scaffolds for LLM-based vulnerability analysis [91,92]. Rather than allowing an LLM to analyze code without constraints, these methods attempt to ground the reasoning process in explicit program relations and independently generated security evidence.

Related studies have also extended LLM-assisted security analysis to adjacent programming environments and codebase-level graph reasoning. VerilogLAVD uses LLM-aided pattern generation for CWE detection in Verilog, illustrating the extension of vulnerability-oriented reasoning to hardware-description languages [93]. Recent codebase-oriented Graph-RAG research further compares AST-derived graphs with LLM-extracted knowledge graphs and highlights how graph provenance and construction reliability may affect downstream code reasoning [94].

Recent neuro-symbolic and agent-based studies further explore the combination of static-analysis rules, graph reasoning, LLM reasoning, verification tools, and iterative decision processes [95,96]. Such systems may perform multiple stages, including suspicious-code retrieval, path analysis, vulnerability explanation, patch generation, and patch verification. This direction is particularly relevant to repository-level vulnerability mining because it can incorporate cross-function, cross-file, cross-module, and dependency-level contexts.

However, graph-guided reasoning does not automatically guarantee correctness. The reasoning result may still be affected by incomplete graph construction, incorrect retrieved evidence, prompt sensitivity, hallucination, or failure of the verification tool. In addition, agent-based pipelines may accumulate errors across multiple stages. Therefore, the outputs of graph-guided systems should be validated using vulnerable lines, dependency paths, patches, test cases, static-analysis reports, or other reproducible security evidence.

6.4. Cost-Benefit and Deployment Analysis

Graph-LLM integration can improve semantic understanding, contextual reasoning, explanation generation, and repair-oriented analysis, but these benefits should be evaluated

together with their computational and engineering costs. Compared with graph-only GNNs, hybrid systems usually require larger pretrained models, additional graph-to-text or graph-to-embedding interfaces, more GPU memory, longer inference time, and more complex deployment pipelines.

Graph-only GNNs remain suitable for large-scale vulnerability screening when program graphs can capture the relevant structural evidence and when low inference latency, stable resource consumption, and reproducible deployment are important. Their main advantages include relatively compact model size, efficient batch processing, explicit structural inductive bias, and easier integration into continuous-integration or static-analysis pipelines. However, their semantic reasoning ability may be limited when vulnerabilities depend on long-range repository context, configuration semantics, natural-language specifications, or complex business logic.

Graph-LLM hybrid methods are more suitable for scenarios that require broader semantic context, natural-language explanations, repository-level reasoning, interactive security analysis, or patch generation. In these settings, program graphs can provide structural constraints and retrieval targets, while LLMs can support semantic interpretation and multi-step reasoning. Nevertheless, the resulting performance gains may not justify the additional cost for routine function-level classification or high-throughput code scanning.

Future studies should therefore report not only predictive performance but also model size, training resources, peak memory usage, inference latency, throughput, energy consumption, and deployment complexity. Comparisons should include graph-only baselines and determine whether the additional semantic capability produces meaningful improvements under realistic security workloads.

A practical deployment strategy may adopt a staged architecture. A lightweight graph-based detector can first perform large-scale screening, after which a Graph-LLM system is selectively applied to high-risk samples for explanation, repository-level investigation, or repair recommendation. Such selective invocation can reduce computational cost while preserving the semantic-reasoning advantages of hybrid models.

Overall, Graph-LLM integration remains an emerging vulnerability-mining paradigm. Its value should not be assessed only by whether it combines larger models with richer graphs, but by whether it provides measurable improvements in detection, localization, explanation, or repair under realistic resource constraints. Reliable Graph-LLM systems should preserve structural grounding, report computational costs, control data leakage, and provide verifiable evidence that can be inspected by security analysts.

7. Cross-Paradigm Discussion and Research Agenda

Existing studies show that GNN-based methods are useful when vulnerability-relevant structures can be explicitly represented in program graphs, such as control flow, data flow, call relations, def-use chains, and smart contract execution logic [16,18,54,75]. However, their reported advantages are strongly affected by dataset quality, graph construction fidelity, evaluation protocols, task granularity, and deployment requirements. Therefore, the key question is not whether graph-based models are useful, but under what conditions their advantages are reliable, explainable, and practically meaningful.

7.1. From Vulnerability Mining Pipeline to Research Problems

GNN-based vulnerability mining should not be understood only as a comparison among model architectures. A practical vulnerability mining pipeline usually involves dataset construction, program representation, model learning, evaluation protocol, and deployment evidence. Each stage introduces different research problems and may limit the effectiveness of later stages. From this perspective, existing graph-based studies should

be analyzed not only by what model they use, but also by what part of the vulnerability mining pipeline they improve and what problems they leave unresolved.

First, dataset construction determines whether models learn real vulnerability mechanisms or benchmark-specific artifacts. Noisy labels, duplicated functions, simplified synthetic samples, and random splits may cause overly optimistic results. Therefore, future datasets should emphasize stricter deduplication, clearer vulnerability labels, and more realistic project-level or repository-level contexts. Second, program representation determines what information is available to the model. ASTs, CFGs, PDGs, CPGs, multi-view graphs, and heterogeneous graphs provide different structural views, but their usefulness depends on whether they preserve vulnerability-relevant control-flow, data-flow, call, and type relations.

Third, model learning determines how structural and semantic information is aggregated. Homogeneous GNNs can introduce structural message passing, but they may mix different program relations in a unified representation space. Heterogeneous GNNs can model relation types more explicitly, but they depend on accurate relation typing and sufficient labeled data. Graph-LLM methods may support semantic reasoning, but their outputs still require graph grounding and evidence verification. Fourth, evaluation determines whether reported performance reflects practical vulnerability mining ability. Function-level random splits are insufficient for assessing cross-project, temporal, cross-dataset, repository-level, or real-world generalization.

Finally, deployment requires actionable evidence rather than only prediction scores. Attention weights, heat maps, and node-importance rankings are not equivalent to verifiable vulnerability evidence. Practical vulnerability mining systems should provide vulnerable lines, dependency paths, patches, static-analysis reports, or other evidence that can support localization, triage, and repair. This pipeline-based view clarifies why many current studies show promising benchmark results but remain difficult to apply in realistic security scenarios. The bottleneck of GNN-based vulnerability mining is not only model expressiveness, but also the alignment among data quality, graph fidelity, task granularity, evaluation protocol, and analyst-facing evidence.

7.2. What Can and Cannot Be Solved by Graph Models

Homogeneous GNNs, heterogeneous GNNs, and Graph-LLM methods address different stages of the above pipeline. Homogeneous GNNs introduce structural message passing into vulnerability detection and are suitable for relatively clear local or intra-procedural graph patterns. Heterogeneous GNNs further model different node types, edge types, and semantic relations, which helps alleviate the semantic-mixing problem of unified graph modeling. Graph-LLM methods introduce semantic reasoning and broader contextual understanding, which may help analyze cross-function or repository-level vulnerability contexts.

However, none of these paradigms can fully solve vulnerability mining alone. Homogeneous GNNs may smooth different program relations into similar representations. Heterogeneous GNNs require accurate relation typing and sufficient labeled data. Graph-LLM methods may suffer from hallucination, high cost, and unverifiable reasoning. More importantly, all three paradigms depend on the quality of program representation and evaluation setting. If vulnerability-relevant dependencies are missing from the graph, or if labels and data splits are unreliable, stronger models may only learn dataset artifacts rather than real vulnerability mechanisms.

7.3. Minimum Requirements for Reliable Vulnerability Benchmarks

Reliable vulnerability benchmarks should evaluate whether a model learns transferable vulnerability mechanisms rather than duplicated code fragments, project-specific conventions, or artifacts introduced during dataset construction. At a minimum, benchmark development and reporting should satisfy the requirements summarized in Table 5.

Table 5. Minimum requirements for reliable software vulnerability benchmarks.

Requirement	Purpose	Recommended Evidence
Exact and near-duplicate removal	Prevent models from memorizing identical or highly similar code fragments across training and test sets.	Report the deduplication method, similarity threshold, duplicate counts, and whether pre-patch and post-patch versions were separated.
Project- or repository-level separation	Reduce leakage caused by project-specific APIs, coding styles, naming conventions, and repeated vulnerability patterns.	Ensure that repositories or projects appearing in the test set are absent from the training and validation sets.
Temporal splitting	Prevent future information, later software versions, or post-disclosure vulnerability patterns from entering the training data.	Split samples according to commit time, release time, patch time, or vulnerability-disclosure time, and publish the temporal boundaries.
Label provenance and verification	Reduce noise introduced by inaccurate vulnerability labels, automatically mined commits, or weak keyword-based labeling.	Provide CVE/CWE mappings, patch links, commit histories, static-analysis evidence, and manual or expert verification procedures.
Repository-level context	Preserve cross-function, cross-file, dependency, configuration, and build-system information that may determine vulnerability behavior.	Release function-level samples together with their repository, call-graph, dependency, and version contexts whenever possible.
Class-distribution transparency	Prevent misleading conclusions caused by hidden class imbalance or inconsistent negative-sample construction.	Report positive and negative sample counts, vulnerability-category distributions, sampling strategies, and class-weighting procedures.
Cross-project and cross-dataset evaluation	Assess whether reported performance generalizes beyond the benchmark distribution used for training.	Report cross-project, cross-version, cross-language, or external-dataset results in addition to random-split results.
Reproducible split metadata	Allow independent verification of leakage control and experimental comparisons.	Release sample identifiers, repository names, commit hashes, split files, preprocessing scripts, and random seeds.

Random function-level splitting should not be treated as sufficient evidence of practical generalization. A benchmark may produce high in-distribution accuracy while still containing duplicated samples, closely related software versions, shared repositories, or future information. Therefore, random-split results should be reported together with stricter project-level, temporal, cross-dataset, or repository-level evaluations.

These requirements do not imply that every benchmark must contain complete repository-scale program analysis. However, the dataset construction process should clearly document which contextual information is preserved, which sources of leakage have been controlled, and which limitations remain. Such transparency is necessary for determining whether performance improvements reflect genuine vulnerability reasoning or benchmark-specific correlations.

7.4. Minimum Requirements for Verifiable Explanations

Explanations in software vulnerability mining should provide evidence that can be independently inspected and related to the actual vulnerability mechanism. Attention weights, heat maps, and node-importance scores may indicate statistical relevance, but

they should not be treated as sufficient evidence of causality. At a minimum, explanation methods should satisfy the requirements summarized in Table 6.

Table 6. Minimum requirements for verifiable vulnerability explanations.

Requirement	Purpose	Possible Evaluation Evidence
Counterfactual faithfulness	Verify that the identified nodes, edges, statements, or paths materially influence the model prediction.	Measure prediction changes after masking, removing, or perturbing the highlighted evidence while preserving unrelated code.
Stability	Ensure that explanations do not change substantially under minor semantics-preserving code transformations.	Compare explanation overlap, ranking correlation, or path consistency after identifier renaming, formatting changes, or equivalent refactoring.
Agreement with security evidence	Determine whether the explanation aligns with independently generated vulnerability evidence.	Compare highlighted statements or paths with static-analysis reports, taint-flow results, security rules, patches, CVE descriptions, or expert annotations.
Code-level localization and path recovery	Provide actionable evidence that supports vulnerability triage and repair.	Evaluate line-level localization, vulnerable-function identification, dependency-path recovery, program-slice overlap, Hit@K, mean average precision, or initial false alarm.
Human auditability and traceability	Allow security analysts to reproduce, inspect, and verify the explanation process.	Provide mappings from graph elements to source-code locations, reproducible evidence logs, model decisions, graph relations, and explanation-generation settings.

A production-oriented explanation should satisfy more than one of these requirements. For example, a node-importance map may be visually intuitive but remains weak evidence if removing the highlighted nodes does not affect the prediction or if the highlighted region does not correspond to a known vulnerable path. Similarly, an explanation that is faithful to the model but unstable under minor code changes may be unsuitable for security auditing.

Future studies should therefore evaluate explanation quality using multiple complementary criteria, including faithfulness, stability, localization accuracy, path consistency, agreement with static-analysis or patch evidence, and analyst-oriented usability. This shift from visualized correlation to verifiable evidence is necessary for supporting vulnerability localization, triage, explanation, and repair.

7.5. Strategic Future Research Agenda

Future research should move beyond incremental model optimization and address the complete vulnerability-mining pipeline. Table 7 summarizes five strategic directions, their technical objectives, and the evidence required to demonstrate practical progress.

First, benchmark development should shift from isolated function-level samples toward repository-level and temporally ordered datasets. Future benchmarks should preserve cross-function, cross-file, dependency, configuration, and build-system contexts while providing transparent deduplication and labeling procedures. Evaluation should include project-level, temporal, cross-version, and cross-dataset settings so that reported improvements can be distinguished from benchmark-specific memorization.

Second, program graph construction should become uncertainty-aware. Existing static analysis tools may omit or incorrectly construct call, data-flow, and dependency relations in the presence of reflection, dynamic dispatch, serialization, framework callbacks, and runtime binding. Future systems should combine static graphs with dynamic traces, points-to analysis, type inference, multi-view representations, and learned edge prediction.

Graph-construction uncertainty should also be propagated to downstream models rather than treating every node and edge as equally reliable.

Table 7. Strategic research agenda for graph-based software vulnerability mining.

Research Direction	Technical Objective	Required Evaluation Evidence
Leakage-resistant repository-level benchmarks	Construct datasets that preserve project, version, dependency, and temporal contexts while controlling duplicate code and future-information leakage.	Deduplication statistics, repository-level splits, temporal boundaries, cross-project testing, external-dataset evaluation, and released split metadata.
Uncertainty-aware high-fidelity graph construction	Improve graph recall and precision under reflection, dynamic dispatch, framework callbacks, incomplete dependencies, and inter-procedural analysis.	Comparison with static and dynamic analysis results, missing-edge and spurious-edge statistics, uncertainty estimates, and downstream sensitivity analysis.
Verifiable analyst-facing explanations	Produce explanations that support localization, dependency-path recovery, triage, and repair rather than only visualizing attention or node importance.	Counterfactual tests, stability analysis, agreement with patches or static-analysis reports, line-level localization, path recovery, and analyst evaluation.
Cost-effective Graph-LLM collaboration	Determine when semantic reasoning from LLMs provides sufficient benefit to justify additional computational and deployment costs.	Graph-only baselines, model size, memory usage, latency, throughput, energy consumption, staged-deployment evaluation, and task-specific performance gains.
Graph-guided autonomous vulnerability repair	Develop constrained agents that integrate graph evidence, static analysis, LLM reasoning, patch generation, and verification tools.	Patch correctness, compilation and test success, vulnerability removal, regression testing, security-rule verification, and prevention of newly introduced defects.

Third, explanation research should focus on analyst-facing and verifiable evidence. Future methods should jointly evaluate counterfactual faithfulness, stability, localization accuracy, path consistency, agreement with security reports, and human usability. Explanations should be connected to source-code lines, dependency paths, program slices, patches, and reproducible analysis logs so that security analysts can inspect how a prediction was produced.

Fourth, Graph-LLM integration should be evaluated according to task-specific cost and benefit. Hybrid systems are likely to be most useful for repository-level investigation, vulnerability explanation, interactive analysis, and repair recommendation, whereas graph-only models may remain more appropriate for large-scale screening. Staged systems that first use lightweight graph models and then invoke LLM-based reasoning for high-risk samples may provide a practical balance between throughput and semantic capability.

Finally, graph-guided autonomous agents represent a promising direction for moving from vulnerability detection to vulnerability repair. Such agents may combine suspicious-code retrieval, graph-based path analysis, static-analysis rules, LLM reasoning, patch generation, test execution, and iterative verification [91,92,95,96]. Program graphs can constrain the agent by identifying affected variables, dependency paths, call relations, and security-sensitive statements, while verification tools can test whether a generated patch removes the vulnerability without introducing functional regressions.

The success of autonomous repair systems should not be measured only by the number of generated patches. Evaluation should determine whether patches compile, pass functional and security tests, preserve intended behavior, remove the original vulnerability, and avoid introducing new defects. Human review should remain part of the deployment process for high-risk software until automated verification becomes sufficiently reliable.

Overall, future progress will depend on coordinated improvements in datasets, graph construction, model design, evaluation, explanation, and repair. The field should therefore move from isolated benchmark optimization toward evidence-grounded systems that

can support reliable vulnerability detection, localization, explanation, and remediation in realistic software repositories.

8. Conclusions

This review examined graph neural networks for software vulnerability mining through five research questions covering program graph representation, homogeneous and heterogeneous GNN architectures, Graph-LLM integration, evaluation reliability, and future research directions. The reviewed evidence shows that program graphs can provide useful structural information for vulnerability detection, localization, explanation, and repair, particularly when vulnerabilities are associated with explicit syntax, control-flow, data-flow, call, or dependency relations.

The effectiveness of graph-based vulnerability mining nevertheless depends strongly on the fidelity of the underlying program representation. Homogeneous GNNs provide efficient structural message passing but may mix different semantic relations, whereas heterogeneous GNNs preserve relation types more explicitly at the cost of greater graph-quality, data, and computational requirements. Graph-LLM systems can further improve semantic reasoning, repository-level analysis, explanation generation, and repair support, but their benefits should be assessed together with model size, memory consumption, inference latency, deployment complexity, and evidence-verification requirements.

The review also indicates that high benchmark performance does not necessarily imply reliable practical capability. Duplicated samples, random function-level splits, noisy labels, temporal leakage, incomplete repository context, and unverifiable explanations may substantially inflate reported results. Reliable future studies should therefore adopt leakage-resistant benchmarks, project-level and temporal evaluation, transparent graph-construction procedures, reproducible split metadata, and explanation protocols based on counterfactual faithfulness, stability, code-level localization, and agreement with independent security evidence.

Future progress should move from isolated model optimization toward pipeline-oriented and evidence-grounded vulnerability analysis. Promising directions include uncertainty-aware graph construction, repository-level evaluation, cost-effective Graph-LLM collaboration, analyst-facing explanations, and graph-guided autonomous agents that integrate vulnerability localization, patch generation, testing, and verification. Overall, graph-based vulnerability mining will become practically valuable only when improvements in data quality, graph fidelity, model design, evaluation rigor, and actionable security evidence are developed together.

Author Contributions: Conceptualization, H.L. and Y.H.; methodology, H.L.; investigation, H.L., X.L. (Xing Li) and L.Z. (Lulu Zeng); resources, Y.H., L.Z. (Lina Zhang), D.Y. and X.L. (Xiaowei Li); writing—original draft preparation, H.L.; writing—review and editing, Y.H., X.L. (Xing Li), L.Z. (Lulu Zeng), L.Z. (Lina Zhang), D.Y. and X.L. (Xiaowei Li); supervision, Y.H.; project administration, Y.H.; funding acquisition, Y.H., L.Z. (Lina Zhang), D.Y. and X.L. (Xiaowei Li). All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Doctoral Research Initiation Fund Project, grant numbers KYBS2021084 and KYBS2023027; the Special Basic Cooperative Research Programs of the Yunnan Provincial Undergraduate Universities' Association, grant numbers 202301BA070001-036 and 202101BA070001-093; the Yunnan Computer Teaching Reform and Research Project, grant number 2024028; the Special Project of the National Natural Science Foundation of China, grant number 62341203; the National Natural Science Foundation of China, grant numbers 32260131 and 62262001; the Yunnan Province Professional Degree Graduate Teaching Case Library Construction Project, grant number 230202011190; the Second Batch of Ideological and Political Education Special Project of Dali University, grant number SZ022025117; the Tenth Batch of Key Teaching Reform Project of Dali

University, grant number JG10102; and the Yunnan Young and Middle-Aged Academic and Technical Leaders Reserve Talent Project in China, grant number 202405AC350023.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new datasets were generated in this study. The literature and materials analyzed in this review are available from the cited publications and public sources.

Acknowledgments: During the preparation of this manuscript, the authors used ChatGPT(GPT-5.5 Thinking, OpenAI) for language editing and formatting assistance. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ladisa, P.; Plate, H.; Martinez, M.; Barais, O. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–25 May 2023; pp. 1509–1526. <https://doi.org/10.1109/SP46215.2023.10179304>.
2. Williams, L.; Benedetti, G.; Hamer, S.; Paramitha, R.; Rahman, I.; Tamanna, M.; Tystahl, G.; Zahan, N.; Morrison, P.; Acar, Y.; et al. Research Directions in Software Supply Chain Security. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 146. <https://doi.org/10.1145/3714464>.
3. Shen, Y.; Gao, X.; Sun, H.; Guo, Y. Understanding Vulnerabilities in Software Supply Chains. *Empir. Softw. Eng.* **2025**, *30*, 20. <https://doi.org/10.1007/s10664-024-10581-2>.
4. Lenarduzzi, V.; Pecorelli, F.; Saarimaki, N.; Lujan, S.; Palomba, F. A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision. *J. Syst. Softw.* **2023**, *198*, 111575. <https://doi.org/10.1016/j.jss.2022.111575>.
5. He, W.; Di, P.; Ming, M.; Zhang, C.; Su, T.; Li, S.; Sui, Y. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *Proc. ACM Softw. Eng.* **2024**, *1*, 1656–1678. <https://doi.org/10.1145/3660781>.
6. Lipp, S.; Banescu, S.; Pretschner, A. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 544–555. <https://doi.org/10.1145/3533767.3534380>.
7. Murali, A.; Mathews, N.S.; Alfadel, M.; Nagappan, M.; Xu, M. FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 778–790. <https://doi.org/10.1145/3597503.3623321>.
8. Hu, H.; Wang, Y.; Rubin, J.; Pradel, M. An Empirical Study of Suppressed Static Analysis Warnings. *Proc. ACM Softw. Eng.* **2025**, *2*, 290–311. <https://doi.org/10.1145/3715729>.
9. Utture, A.; Liu, S.; Kalhauge, C.G.; Palsberg, J. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 2043–2055. <https://doi.org/10.1145/3510003.3510166>.
10. Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Gall, H.C.; Zaidman, A. How Developers Engage with Static Analysis Tools in Different Contexts. *Empir. Softw. Eng.* **2020**, *25*, 1419–1457. <https://doi.org/10.1007/s10664-019-09750-5>.
11. Guo, Z.; Tan, T.; Liu, S.; Liu, X.; Lai, W.; Yang, Y.; Li, Y.; Chen, L.; Dong, W.; Zhou, Y. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Trans. Softw. Eng.* **2023**, *49*, 5154–5188. <https://doi.org/10.1109/TSE.2023.3329667>.
12. Hellendoorn, V.J.; Maniatis, P.; Singh, R.; Sutton, C.; Bieber, D. Global Relational Models of Source Code. In Proceedings of the International Conference on Learning Representations (ICLR), Virtual Event, 26 April–1 May 2020.
13. Bieber, D.; Sutton, C.A.; Larochelle, H.; Tarlow, D. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2020; Volume 33, pp. 8626–8637.
14. Liu, J.; Zeng, J.; Wang, X.; Liang, Z. Learning Graph-Based Code Representations for Source-Level Functional Similarity Detection. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 345–357. <https://doi.org/10.1109/ICSE48619.2023.00040>.
15. Long, T.; Xie, Y.; Chen, X.; Zhang, W.; Cao, Q.; Yu, Y. Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection. *Proc. AAAI Conf. Artif. Intell.* **2022**, *36*, 5792–5799. <https://doi.org/10.1609/aaai.v36i5.20522>.

16. Nguyen, V.-A.; Nguyen, D.Q.; Nguyen, V.; Le, T.; Tran, Q.H.; Phung, D. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In Proceedings of the 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 2022), Pittsburgh, PA, USA, 21–29 May 2022. <https://doi.org/10.1145/3510454.3516865>.
17. Luo, Y.; Xu, W.; Xu, D. Compact Abstract Graphs for Detecting Code Vulnerability with GNN Models. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 497–507. <https://doi.org/10.1145/3564625.3564655>.
18. Qiu, F.; Liu, Z.; Hu, X.; Xia, X.; Chen, G.; Wang, X. Vulnerability Detection via Multiple-Graph-Based Code Representation. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2178–2199. <https://doi.org/10.1109/TSE.2024.3427815>.
19. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart Contract Vulnerability Detection Using Graph Neural Network. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI), Yokohama, Japan, 11–17 July 2020; pp. 3283–3290. <https://doi.org/10.24963/ijcai.2020/454>.
20. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Softw. Eng.* **2022**, *48*, 3280–3296. <https://doi.org/10.1109/TSE.2021.3087402>.
21. Risse, N.; Liu, J.; Böhme, M. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* **2025**, *2*, 388–410. <https://doi.org/10.1145/3728887>.
22. Wang, X.; Hu, R.; Gao, C.; Wen, X.-C.; Chen, Y.; Liao, Q. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Lisbon, Portugal, 14–20 April 2024; pp. 472–483. <https://doi.org/10.1145/3639478.3647634>.
23. Reif, M.; Kübler, F.; Eichberg, M.; Mezini, M. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In Proceedings of the Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, Amsterdam, The Netherlands, 16–21 July 2018; pp. 107–112. <https://doi.org/10.1145/3236454.3236503>.
24. Sui, L.; Dietrich, J.; Tahir, A.; Fourtounis, G. On the Recall of Static Call Graph Construction in Practice. In Proceedings of the 42nd International Conference on Software Engineering (ICSE), Seoul, Republic of Korea, 27 June–19 July 2020; pp. 1049–1060. <https://doi.org/10.1145/3377811.3380441>.
25. Li, Y.; Tan, T.; Xue, J. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*, 7. <https://doi.org/10.1145/3295739>.
26. Ali, K.; Lai, X.; Luo, Z.; Lhoták, O.; Dolby, J.; Tip, F. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2644–2666. <https://doi.org/10.1109/TSE.2019.2956925>.
27. Helm, D.; Keidel, S.; Kampkötter, A.; Düsing, J.; Roth, T.; Hermann, B.; Mezini, M. Total Recall? How Good Are Static Call Graphs Really? In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, 16–20 September 2024; pp. 112–123. <https://doi.org/10.1145/3650212.3652114>.
28. Luo, F.; Luo, R.; Chen, T.; Qiao, A.; He, Z.; Song, S.; Jiang, Y.; Li, S. SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024; pp. 2098–2110. <https://doi.org/10.1145/3597503.3639213>.
29. Wen, X.-C.; Gao, C.; Ye, J.; Li, Y.; Tian, Z.; Jia, Y.; Wang, X. Meta-Path Based Attentional Graph Learning Model for Vulnerability Detection. *IEEE Trans. Softw. Eng.* **2024**, *50*, 360–375. <https://doi.org/10.1109/TSE.2023.3340267>.
30. Nguyen, H.H.; Nguyen, N.-M.; Xie, C.; Ahmadi, Z.; Kudenko, D.; Doan, T.-N.; Jiang, L. MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. In Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR), Melbourne, Australia, 15–16 May 2023; pp. 334–346. <https://doi.org/10.1109/MSR59073.2023.00052>.
31. Zhang, J.; Lu, G.; Yu, J. A Smart Contract Vulnerability Detection Method Based on Heterogeneous Contract Semantic Graphs and Pre-Training Techniques. *Electronics* **2024**, *13*, 3786. <https://doi.org/10.3390/electronics13183786>.
32. Gu, X.; Duan, X.; Ren, S.; Zhang, J.; Cui, Z. Detecting Smart Contract Vulnerabilities Based on Fusing Semantic and Syntax Structure Information. In Proceedings of the 2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), Kaifeng, China, 28 October–1 November 2024; pp. 1982–1987. <https://doi.org/10.1109/ISPA63168.2024.00270>.
33. Ghaffarian, S.M.; Shahriari, H.R. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* **2017**, *50*, 56. <https://doi.org/10.1145/3092566>.
34. Lin, G.; Wen, S.; Han, Q.-L.; Zhang, J.; Xiang, Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. <https://doi.org/10.1109/JPROC.2020.2993293>.
35. Shiri Harzevili, N.; Boaye Belle, A.; Wang, J.; Wang, S.; Jiang, Z.M.; Nagappan, N. A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Comput. Surv.* **2025**, *57*, 55. <https://doi.org/10.1145/3699711>.
36. Chu, H.; Zhang, P.; Dong, H.; Xiao, Y.; Ji, S.; Li, W. A Survey on Smart Contract Vulnerabilities: Data Sources, Detection and Repair. *Inf. Softw. Technol.* **2023**, *159*, 107221. <https://doi.org/10.1016/j.infsof.2023.107221>.
37. Sheng, Z.; Chen, Z.; Gu, S.; Huang, H.; Gu, G.; Huang, J. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *ACM Comput. Surv.* **2026**, *58*, 134. <https://doi.org/10.1145/3769082>.

38. Page, M.J.; McKenzie, J.E.; Bossuyt, P.M.; Boutron, I.; Hoffmann, T.C.; Mulrow, C.D.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews. *BMJ* **2021**, 372, n71. <https://doi.org/10.1136/bmj.n71>.
39. Page, M.J.; Moher, D.; Bossuyt, P.M.; Boutron, I.; Hoffmann, T.C.; Mulrow, C.D.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. PRISMA 2020 Explanation and Elaboration: Updated Guidance and Exemplars for Reporting Systematic Reviews. *BMJ* **2021**, 372, n160. <https://doi.org/10.1136/bmj.n160>.
40. Grant, M.J.; Booth, A. A Typology of Reviews: An Analysis of 14 Review Types and Associated Methodologies. *Health Inf. Libr. J.* **2009**, 26, 91–108. <https://doi.org/10.1111/j.1471-1842.2009.00848.x>.
41. Snyder, H. Literature Review as a Research Methodology: An Overview and Guidelines. *J. Bus. Res.* **2019**, 104, 333–339. <https://doi.org/10.1016/j.jbusres.2019.07.039>.
42. Vagavolu, D.; Swarna, K.C.; Chimalakonda, S. A Mocktail of Source Code Representations. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021; pp. 1296–1300. <https://doi.org/10.1109/ASE51524.2021.9678551>.
43. Fang, C.; Shen, Y.; Huang, J.; Shi, Q. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual Event, 18–22 July 2020; pp. 516–527. <https://doi.org/10.1145/3395363.3397362>.
44. Ramu, R.; Upadhyaya, G.; Nguyen, H.A.; Rajan, H. Hybrid Traversal: Efficient Source Code Analysis at Scale. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Gothenburg, Sweden, 27 May–3 June 2018; pp. 412–413. <https://doi.org/10.1145/3183440.3195033>.
45. Li, S.; Kang, M.; Hou, J.; Cao, Y. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; USENIX Association: Berkeley, CA, USA, 2022; pp. 143–160.
46. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 590–604. <https://doi.org/10.1109/SP.2014.44>.
47. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, 19, 2244–2258. <https://doi.org/10.1109/TDSC.2021.3051525>.
48. Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; Jin, H. VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector. *IEEE Trans. Dependable Secur. Comput.* **2022**, 19, 2821–2837. <https://doi.org/10.1109/TDSC.2021.3076142>.
49. Wu, T.; Chen, L.; Du, G.; Meng, D.; Shi, G. UltraVCS: Ultra-Fine-Grained Variable-Based Code Slicing for Automated Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* **2024**, 19, 3986–4000. <https://doi.org/10.1109/TIFS.2024.3374219>.
50. Shezan, F.H.; Su, Z.; Kang, M.; Phair, N.; Thomas, P.W.; Van Dam, M.; Cao, Y.; Tian, Y. CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-Language Code Property Graph. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 27 February–3 March 2023. <https://doi.org/10.14722/ndss.2023.24610>.
51. Qin, K.; Ye, Z.; Wang, Z.; Li, W.; Zhou, L.; Zhang, C.; Song, D.; Gervais, A. Enhancing Smart Contract Security Analysis with Execution Property Graphs. *Proc. ACM Softw. Eng.* **2025**, 2, 1101–1122. <https://doi.org/10.1145/3728924>.
52. Wi, S.; Woo, S.; Whang, J.J.; Son, S. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In Proceedings of the ACM Web Conference 2022 (WWW), Lyon, France, 25–29 April 2022; pp. 755–766. <https://doi.org/10.1145/3485447.3512235>.
53. Swarna, K.C.; Mathews, N.S.; Vagavolu, D.; Chimalakonda, S. On the Impact of Multiple Source Code Representations on Software Engineering Tasks: An Empirical Study. *J. Syst. Softw.* **2024**, 210, 111941. <https://doi.org/10.1016/j.jss.2023.111941>.
54. Cheng, X.; Wang, H.; Hua, J.; Xu, G.; Sui, Y. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* **2021**, 30, 38. <https://doi.org/10.1145/3436877>.
55. Xiao, P.; Xiao, Q.; Zhang, X.; Wang, Y.; Yang, F. Vulnerability Detection Based on Enhanced Graph Representation Learning. *IEEE Trans. Inf. Forensics Secur.* **2024**, 19, 5120–5135. <https://doi.org/10.1109/TIFS.2024.3392536>.
56. Huang, Y.; He, M.; Wang, X.; Zhang, J. HeVulD: A Static Vulnerability Detection Method Using Heterogeneous Graph Code Representation. *IEEE Trans. Inf. Forensics Secur.* **2024**, 19, 9129–9144. <https://doi.org/10.1109/TIFS.2024.3457162>.
57. Thakur, M. How (Not) to Write Java Pointer Analyses after 2020. In Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Virtual Event, 18–23 November 2020; pp. 134–145. <https://doi.org/10.1145/3426428.3426923>.
58. Wimmer, C.; Stancu, C.; Kozak, D.; Würthinger, T. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* **2024**, 8, 990–1013. <https://doi.org/10.1145/3656417>.
59. Santos, J.C.S.; Jones, R.A.; Ashiogwu, C.; Mirakhorli, M. Serialization-Aware Call Graph Construction. In Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP), Virtual Event, 22 June 2021; pp. 37–42. <https://doi.org/10.1145/3460946.3464319>.

60. Santos, J.C.S.; Mirakhorli, M.; Shokri, A. Seneca: Taint-Based Call Graph Construction for Java Object Deserialization. *Proc. ACM Program. Lang.* **2024**, *8*, 1125–1153. <https://doi.org/10.1145/3649851>.
61. Samhi, J.; Miltenberger, M.; Alecci, M.; Arzt, S.; Bissyandé, T.F.D.A.; Klein, J. Do You Have 5 Min? Improving Call Graph Analysis with Runtime Information. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering: Companion Proceedings (FSE Companion)*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 540–544. <https://doi.org/10.1145/3696630.3728495>.
62. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-Training Code Representations with Data Flow. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Virtual, 3–7 May 2021.
63. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; Zhou, M. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>.
64. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C.H. CodeT5: Identifier-Aware Unified Pre-Trained Encoder–Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*; Association for Computational Linguistics: Punta Cana, Dominican Republic, 2021; pp. 8696–8708. <https://doi.org/10.18653/v1/2021.emnlp-main.685>.
65. Ahmad, W.U.; Chakraborty, S.; Ray, B.; Chang, K.-W. Unified Pre-Training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021; pp. 2655–2668. <https://doi.org/10.18653/v1/2021.naacl-main.211>.
66. Du, Y.; Yu, Z. Pre-Training Code Representation with Semantic Flow Graph for Effective Bug Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, CA, USA, 3–9 December 2023; pp. 579–591. <https://doi.org/10.1145/3611643.3616338>.
67. Nguyen, S.; Nguyen, T.-T.; Vu, T.T.; Do, T.-D.; Ngo, K.-T.; Vo, H.D. Code-Centric Learning-Based Just-in-Time Vulnerability Detection. *J. Syst. Softw.* **2024**, *214*, 112014. <https://doi.org/10.1016/j.jss.2024.112014>.
68. Cao, S.; Sun, X.; Wu, X.; Lo, D.; Bo, L.; Li, B.; Liu, W. Coca: Improving and Explaining Graph Neural Network-Based Vulnerability Detection Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, 14–20 April 2024; pp. 1911–1923. <https://doi.org/10.1145/3597503.3639168>.
69. Peng, X.; Wang, S.; Qin, Y.; Lin, B.; Chen, L.; Cheng, J.; Mao, X. Keep It Simple: Self-Adaptive Code Graph Simplification for Accurate Vulnerability Detection. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2744–2763. <https://doi.org/10.1109/TSE.2025.3593515>.
70. Xu, C.; Xu, H.; Zhu, L.; Shen, X.; Sharif, K. Enhanced Smart Contract Vulnerability Detection via Graph Neural Networks: Achieving High Accuracy and Efficiency. *IEEE Trans. Softw. Eng.* **2025**, *51*, 1854–1865. <https://doi.org/10.1109/TSE.2025.3570421>.
71. Wu, B.; Liu, S.; Xiao, Y.; Li, Z.; Sun, J.; Lin, S.-W. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-Procedural Slicing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, CA, USA, 3–9 December 2023; pp. 1371–1383. <https://doi.org/10.1145/3611643.3616351>.
72. Zeng, Q.; Xiong, D.; Wu, Z.; Qian, K.; Wang, Y.; Su, Y. TACSan: Enhancing Vulnerability Detection with Graph Neural Network. *Electronics* **2024**, *13*, 3813. <https://doi.org/10.3390/electronics13193813>.
73. Chu, Z.; Wan, Y.; Li, Q.; Wu, Y.; Zhang, H.; Sui, Y.; Xu, G.; Jin, H. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Vienna, Austria, 16–20 September 2024; pp. 389–401. <https://doi.org/10.1145/3650212.3652136>.
74. Hu, Y.; Wang, S.; Li, W.; Peng, J.; Wu, Y.; Zou, D.; Jin, H. Interpreters for GNN-Based Vulnerability Detection: Are We There Yet? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, 17–21 July 2023; pp. 1407–1419. <https://doi.org/10.1145/3597926.3598145>.
75. Zhang, C.; Liu, B.; Xin, Y.; Yao, L. CPVD: Cross-Project Vulnerability Detection Based on Graph Attention Network and Domain Adaptation. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4152–4168. <https://doi.org/10.1109/TSE.2023.3285910>.
76. Li, M.; Zhu, Y.; Liu, Y.; Li, Z.; Huang, J. Graph Attention Network Vulnerability Detection Model with Global Feature Augmentation for Smart Contracts. *J. Cloud Comput.* **2025**, *14*, 67. <https://doi.org/10.1186/s13677-025-00798-x>.
77. Cao, S.; Sun, X.; Wu, X.; Lo, D.; Bo, L.; Li, B.; Liu, X.; Lin, X.; Liu, W. Snopy: Bridging Sample Denoising with Causal Graph Learning for Effective Vulnerability Detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sacramento, CA, USA, 27 October–1 November 2024; pp. 606–618. <https://doi.org/10.1145/3691620.3695057>.

78. Wang, X.; Ji, H.; Shi, C.; Wang, B.; Ye, Y.; Cui, P.; Yu, P.S. Heterogeneous Graph Attention Network. In Proceedings of the World Wide Web Conference (WWW), San Francisco, CA, USA, 13–17 May 2019; pp. 2022–2032. <https://doi.org/10.1145/3308558.3313562>.
79. Zhang, Y.; Ju, X.; Chen, X.; Alam, M.; Ren, Z. HGAN4VD: Leveraging Heterogeneous Graph Attention Networks for Enhanced Vulnerability Detection. *Comput. Secur.* **2025**, *157*, 104548. <https://doi.org/10.1016/j.cose.2025.104548>.
80. Nguyen, H.H.; Nguyen, N.-M.; Xie, C.; Ahmadi, Z.; Kudenko, D.; Doan, T.-N.; Jiang, L. MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities. In Proceedings of the 9th IEEE International Conference on Data Science and Advanced Analytics (DSAA), Shenzhen, China, 13–16 October 2022; pp. 1–10. <https://doi.org/10.1109/DSAA54385.2022.10032337>.
81. Li, H.; Xiong, G.; Hou, C.; Gou, G.; Chen, Z.; Li, Z. Smart Contract Vulnerability Detection Based on AST-Augmented Heterogeneous Graphs. In Proceedings of the 43rd IEEE International Performance, Computing, and Communications Conference (IPCCC), Orlando, FL, USA, 22–24 November 2024; pp. 1–10. <https://doi.org/10.1109/IPCCC59868.2024.10850061>.
82. Schlichtkrull, M.; Kipf, T.N.; Bloem, P.; van den Berg, R.; Titov, I.; Welling, M. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web, Proceedings of the 15th International Conference, ESWC 2018*; Springer: Cham, Switzerland, 2018; Volume 10843, pp. 593–607. https://doi.org/10.1007/978-3-319-93417-4_38.
83. Yang, H.; Yang, H.; Zhang, L.; Cheng, X. Source Code Vulnerability Detection Using Vulnerability Dependency Representation Graph. In Proceedings of the 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Wuhan, China, 9–11 December 2022; pp. 457–464. <https://doi.org/10.1109/TrustCom56396.2022.00070>.
84. Hu, Z.; Dong, Y.; Wang, K.; Sun, Y. Heterogeneous Graph Transformer. In Proceedings of the Web Conference 2020 (WWW), Taipei, Taiwan, 20–24 April 2020; pp. 2704–2710. <https://doi.org/10.1145/3366423.3380027>.
85. Lu, G.; Ju, X.; Chen, X.; Pei, W.; Cai, Z. GRACE: Empowering LLM-Based Software Vulnerability Detection with Graph Structure and In-Context Learning. *J. Syst. Softw.* **2024**, *212*, 112031. <https://doi.org/10.1016/j.jss.2024.112031>.
86. Li, J.; Cui, L.; Zhao, S.; Yang, Y.; Li, L.; Zhu, H. CLVeR: Multi-Modal Contrastive Learning for Vulnerability Code Representation. In *Findings of the Association for Computational Linguistics: ACL 2025*; Association for Computational Linguistics: Vienna, Austria, 2025; pp. 7940–7951. <https://doi.org/10.18653/v1/2025.findings-acl.414>.
87. Feng, R.; Pearce, H.; Liguori, P.; Sui, Y. CGP-Tuning: Structure-Aware Soft Prompt Tuning for Code Vulnerability Detection. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2533–2548. <https://doi.org/10.1109/TSE.2025.3591934>.
88. Lekssays, A.; Mouhcine, H.; Tran, K.; Yu, T.; Khalil, I. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; USENIX Association: Berkeley, CA, USA, 2025; pp. 489–507.
89. Liu, R.; Wang, Y.; Xu, H.; Sun, J.; Zhang, F.; Li, P.; Guo, Z. Vul-LMGNNs: Fusing Language Models and Online-Distilled Graph Neural Networks for Code Vulnerability Detection. *Inf. Fusion* **2025**, *115*, 102748. <https://doi.org/10.1016/j.inffus.2024.102748>.
90. Nguyen, N.-M.; Nguyen, H.H.; Le Thanh, L.; Ahmadi, Z.; Doan, T.-N.; Wu, D.; Jiang, L. MANDO-LLM: Heterogeneous Graph Transformers with Large Language Models for Smart Contract Vulnerability Detection. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 144. <https://doi.org/10.1145/3765751>.
91. Du, X.; Zheng, G.; Wang, K.; Feng, J.; Deng, W.; Liu, M.; Chen, B.; Peng, X.; Ma, T.; Lou, Y. Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG. *arXiv* **2024**, arXiv:2406.11147. <https://doi.org/10.48550/arXiv.2406.11147>.
92. Li, Z.; Dutta, S.; Naik, M. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In Proceedings of the International Conference on Learning Representations (ICLR), Singapore, 24–28 April 2025.
93. Long, X.; Xia, Y.; Kuang, L.; Wan, Y.; Liu, Z. VerilogLAVD: LLM-Aided Pattern Generation for Verilog CWE Detection. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL 2026)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2026; pp. 28292–28308. <https://doi.org/10.18653/v1/2026.acl-long.1304>.
94. Chinthareddy, M.R. Reliable Graph-RAG for Codebases: AST-Derived Graphs vs. LLM-Extracted Knowledge Graphs. *arXiv* **2026**, arXiv:2601.08773. <https://doi.org/10.48550/arXiv.2601.08773>.
95. Li, P.; Yao, S.; Korich, J.S.; Luo, C.; Yu, J.; Cao, Y.; Yang, J. Automated Static Vulnerability Detection via a Holistic Neuro-Symbolic Approach. *arXiv* **2025**, arXiv:2504.16057. <https://doi.org/10.48550/arXiv.2504.16057>.
96. Nie, Y.; Li, H.; Guo, C.; Jiang, R.; Wang, Z.; Li, B.; Song, D.; Guo, W. VulnLLM-R: Specialized Reasoning LLM with Agent Scaffold for Vulnerability Detection. *arXiv* **2025**, arXiv:2512.07533. <https://doi.org/10.48550/arXiv.2512.07533>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.