



Review

Program Graph Learning for Software Vulnerability Analysis: A Survey

Tong Yu ^{1,†}, Junjie Wang ^{1,†}, Ming Li ², Yuhang Hu ¹, Junwei Hu ¹, Xiantao Cai ¹, Wenbin Hu ^{1,3,*} and Alessio Borgi ^{4,5}

¹ School of Computer Science, Wuhan University, Wuhan 430072, China

² Zhejiang Key Laboratory of Intelligent Education Technology and Application, Zhejiang Normal University, Jinhua 321017, China

³ Wuhan University Shenzhen Research Institute, Shenzhen 518057, China

⁴ Department of Computer Science and Technology, University of Cambridge, Cambridge CB3 0FD, UK

⁵ Department of Computer, Control and Management Engineering, Sapienza University of Rome, 00185 Rome, Italy

* Correspondence: hwb@whu.edu.cn

† These authors contributed equally to this work.

How To Cite: Yu, T.; Wang, J.; Li, M.; et al. Program Graph Learning for Software Vulnerability Analysis: A Survey. *Transactions on Graph Intelligence and Network Applications* **2026**.

Received: 9 July 2026

Revised: 29 August 2026

Accepted: 1 September 2026

Published: 9 September 2026

Abstract: Software vulnerabilities represent an enduring threat to modern cyberspace. Effective vulnerability detection increasingly relies on reasoning about complex program semantics, structural dependencies, and execution behaviors. Consequently, extracting vulnerability-relevant features from code efficiently has become a pressing research issue. In recent years, advances in graph representation learning and large language models have reframed vulnerability analysis as a graph learning problem over program entities. By building on and extending traditional methods including static analysis, dynamic analysis, symbolic execution, and fuzzing, this perspective enables more effective vulnerability detection, fine-grained localization, multi-class classification, and repair. We review recent progress in graph-driven vulnerability intelligence, structuring the survey around these tasks and then providing a structured overview of representative graph-based methods, graph-based multimodal methods, LLM-assisted methods, and vulnerability repair methods, together with widely adopted datasets, program graph construction tools, and computing frameworks. We then highlight several persistent challenges: interpretability, cross-language generalization, data quality and label noise, the lack of benchmark standardization, and agent-oriented vulnerability analysis. These challenges are often intertwined, complicating piecemeal solutions. Addressing them will be critical for developing reliable, transferable, and explainable vulnerability intelligence systems.

Keywords: graph representation learning; software vulnerability detection; program graph

1. Introduction

Software vulnerabilities constitute a critical threat to modern cyberspace, enabling remote code execution, privilege escalation, data leakage, and large-scale compromise [1–4]. The economic toll is substantial: the global average cost of a data breach reached USD 4.88 million in 2024 [5]; an analysis of 348 open-source smart contracts indicated that their maximum total value locked exceeded USD 11.2 billion [6]; and Cybersecurity Ventures projected that global cybercrime damages would reach USD 10.5 trillion annually by 2025 [7]. The widespread reliance on open-source components and automated software supply chains further amplifies these risks, as a single vulnerability can propagate across dependency networks and magnify both the attack surface and remediation cost [8–10]. Vulnerability detection thus transcends a mere code-quality concern; it is a foundational capability for preventing exploitation, reducing economic losses, and mitigating systemic supply-chain risk [11–13]. Yet the gap between detecting a vulnerability and acting on that information remains substantial.



Copyright: © 2026 by the authors. This is an open access article under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Publisher's Note: Scilight stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Traditional vulnerability detection methods, including static analysis [14, 15], dynamic analysis [16, 17], symbolic execution [18, 19], and fuzzing [20–22], suffer from well-known limitations: high false positive rates, incomplete path coverage, limited ability to model complex program semantics, and insufficient inter-procedural reasoning. As a result, they often miss deep logical flaws and vulnerabilities that hinge on specific semantic conditions [23, 24]. To move beyond these constraints, researchers have increasingly adopted graph-based representations that abstract source code into structures capturing program structure, semantics, and execution behaviours, making program graph-based vulnerability detection a prominent research direction [25, 26]. Program graphs explicitly encode control flow, data flow, call relations, and syntactic dependencies, yielding representations that mirror program execution behaviour and information propagation paths more faithfully than flat token sequences [27, 28]. Different graph representations, such as abstract syntax trees (ASTs), control-flow graphs (CFGs), data-flow graphs (DFGs), program dependence graphs (PDGs), and code property graphs (CPGs), capture distinct types of program information and offer complementary strengths for vulnerability detection [29]. More importantly, by modeling structural dependencies among program entities, program graphs directly support vulnerability-oriented semantic reasoning: control-flow and data-flow edges reveal execution constraints and value propagation, while call relations and slicing-based representations preserve inter-procedural context that is often critical for identifying vulnerable behaviours. Recent work demonstrates that graph- and slice-based representations improve vulnerability detection by integrating multiple semantic views of code and can further enable fine-grained localization by tracing vulnerable lines or dependency paths [27, 30–32]. In effect, program graphs offer a key abstraction that shifts vulnerability analysis from shallow pattern recognition toward structured semantic modeling and reasoning about unsafe contexts, attacker-controllable data flows, and vulnerability-relevant execution dependencies. As shown in Figure 1, a typical graph-driven vulnerability detection pipeline first transforms source code and datasets into program graphs, then assigns feature embeddings to graph nodes, applies GNN-based message passing to propagate and aggregate structural and semantic information, and finally adapts the learned graph representations to downstream tasks.

Vulnerability detection alone is rarely sufficient to deliver adequate security assurance in practice: a binary prediction may signal the possible existence of a vulnerability but cannot explain where the vulnerable behavior originates, why the vulnerability is triggered, or what type of weakness is involved [33–36]. Security analysts need evidence that links model predictions to concrete program elements—vulnerable lines, missing security checks, source-to-sink propagation paths, and inter-procedural calling contexts. Without such evidence, detection results remain difficult to triage, reproduce, and translate into effective patches. Recognizing this, recent work in automated vulnerability analysis is increasingly treating vulnerability handling as a multi-stage pipeline that goes beyond detection, further encompassing vulnerability localization, classification, patch generation, and patch validation [37–40]. Studies on vulnerability localization and explanation have further shown that fine-grained representations, feature fusion, and dependency-aware reasoning can help identify vulnerable lines and provide actionable evidence for repair [34, 41]. This shift is particularly relevant for graph-driven methods, because program graphs can expose not only suspicious code regions but also the structural and semantic dependencies that make them exploitable. Building on this observation, we adopt a broader perspective: we treat vulnerability detection as the entry point of a vulnerability intelligence framework that integrates detection, classification, localization, and repair, aiming to support the full reasoning process from identifying vulnerable code to understanding and mitigating security risks.

Motivated by these observations, we provide a comprehensive and systematic review of graph-driven vulnerability intelligence. Although graph-based learning methods have been widely applied to software vulnerability detection, existing studies remain relatively fragmented. Most works focus on isolated prediction tasks, while relatively few studies systematically examine the field from the perspective of a complete vulnerability analysis workflow. To advance the field, we aim to provide an in-depth summary and analysis that spans task formulation, representative detection methods, program graph construction tools, benchmark datasets, computing frameworks, and current challenges and opportunities [24, 27, 28, 34, 39, 42].

In this survey, we do not treat vulnerability detection merely as an isolated binary classification problem. Instead, we regard it as the entry point of a broader graph-driven vulnerability intelligence pipeline. In this pipeline, program graphs serve as the central abstraction that organically connects source code representation, program graph construction, graph preprocessing and slicing, semantic dependency modeling, graph neural reasoning, code-text representation, large language model (LLM)-based analysis, and retrieval-augmented generation (RAG)-based knowledge integration. From this perspective, vulnerability detection is not limited to determining whether a function or code fragment is vulnerable, but should further support downstream tasks such as vulnerability localization, vulnerability type classification, patch generation, and repair validation. Consequently, this survey examines not only detection performance, but also whether existing methods can capture vulnerability-relevant control-flow, data-flow,

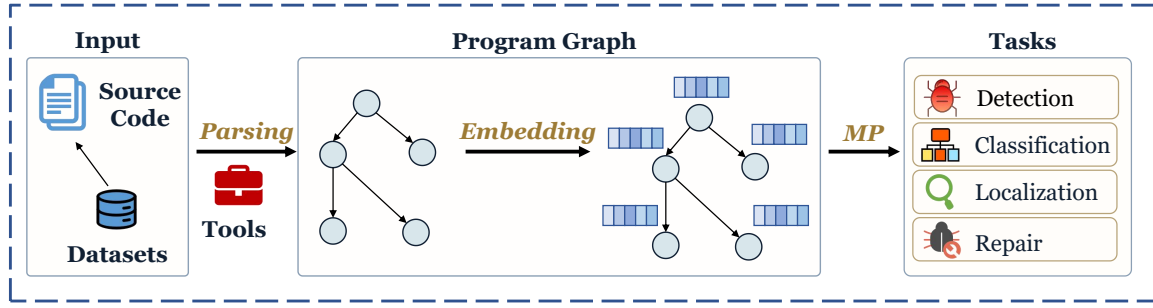


Figure 1. A generic graph-learning pipeline for software vulnerability analysis. MP denotes message passing, and embedding denotes the process of assigning feature representations to graph nodes.

call, and program-dependence relations; integrate lexical code information, historical vulnerability records, and external security knowledge; provide faithful security evidence; and ultimately support scalable, interpretable, and evidence-grounded vulnerability analysis in real-world software security scenarios.

The rest of this paper is organized as follows. Section 2 formulates vulnerability intelligence as a set of graph-driven tasks, including vulnerability detection, vulnerability localization, vulnerability classification, and vulnerability repair. Section 3 reviews representative vulnerability detection and analysis methods, covering graph-based methods, graph-based multimodal methods, LLM-assisted methods, and vulnerability repair methods. Section 4 summarizes representative datasets, program graph construction tools, and computing frameworks that support graph-driven vulnerability analysis. Section 5 discusses the key challenges and opportunities in this field, including poor interpretability, limited generalization, label noise, benchmark inconsistency, and emerging agent-related security risks. Section 6 concludes this survey and outlines future research directions.

2. Problem Definition

We formalize vulnerability analysis as a graph learning problem over program entities. Given a program entity P (e.g., a function, file, or project), our aim is to construct a graph that encodes vulnerability-relevant semantics and to learn mappings from this graph to vulnerability labels, locations, or types.

2.1. Program Graph Representation

We represent P as a typed program graph $\mathcal{G}_P = (V, \{E_r\}_{r \in R}, X, R)$, where V is the set of program elements, X denotes the node feature matrix, and R is the set of relation types. For each relation $r \in R$, $E_r \subseteq V \times V$ denotes the corresponding typed edge set, and the adjacency matrix induced by E_r is denoted by A_r . Depending on the program representation, R may include syntactic, control-flow, data-flow, call, control-dependence, and data-dependence relations. Accordingly, $\mathcal{G}_P$ may represent a function, file, or project, with call and interprocedural data-flow or dependence relations connecting program elements across function boundaries.

2.2. Vulnerability Detection and Classification

We formulate vulnerability detection and classification as graph-level prediction. Given $\mathcal{G}_P$, represented by X and $\{A_r\}_{r \in R}$, the model predicts a label:

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

where $c = 0$ denotes a benign entity and $c \in \{1, \dots, K\}$ denotes a specific vulnerability type, typically defined by a Common Weakness Enumeration (CWE) category. Binary vulnerability detection can be regarded as a special case of this formulation by collapsing all vulnerability types into a single positive class:

$$y = \mathbb{I}[c > 0], \quad y \in \{0, 1\},$$

where $y = 1$ indicates that P is vulnerable and $y = 0$ indicates that it is benign.

2.3. Vulnerability Localization

Vulnerability localization aims to identify the fine-grained source-code regions associated with a vulnerability. In this paper, we use physical source-code lines as the basic localization unit and therefore formulate vulnerability localization as a line-level prediction task. Given $\mathcal{G}_P$, the model predicts a binary label sequence:

$$g_\theta(\mathcal{G}_P) \rightarrow \mathbf{s}, \quad \mathbf{s} \in \{0, 1\}^n,$$

where n denotes the number of physical source-code lines in program P . For each source-code line i , $s_i = 1$ indicates that the line is vulnerability-related, whereas $s_i = 0$ indicates that it is benign. Equivalently, the localization result can be represented as the set of vulnerable source-code lines:

$$S = \{i \mid s_i = 1\}.$$

2.4. Vulnerability Repair

We formalize vulnerability repair as graph-conditioned patch generation. Given a vulnerable program graph $\mathcal{G}_P$, together with optional auxiliary outputs (e.g., the vulnerability type c and the set of localized vulnerable lines S), the model predicts a patch:

$$r_\theta(\mathcal{G}_P, c, S) \rightarrow \Delta,$$

where Δ denotes a sequence (or a set, depending on the formulation) of edit operations over the original program entity P . Applying these operations yields the repaired program:

$$P' = \text{Apply}(P, \Delta),$$

where $\text{Apply}(P, \Delta)$ denotes applying the patch Δ to the original program P . A correct patch should eliminate the vulnerability while preserving the intended functionality of the original program.

In graph-based repair, Δ can be interpreted as edits over program elements or graph nodes: inserting missing checks, replacing unsafe API calls, modifying control predicates, or sanitizing attacker-controllable data flows. In this way, vulnerability repair extends localization beyond identifying vulnerable lines; it generates concrete transformations that break vulnerability-triggering dependencies.

3. Approaches for Vulnerability Detection and Analysis

We classify automated vulnerability detection and analysis methods into four categories: graph-based methods, which explicitly capture program structures; graph-based multimodal methods, which integrate diverse vulnerability-relevant modalities; LLM-assisted methods, which leverage large language models for code understanding and reasoning; and vulnerability repair methods, which further generate patches to remove vulnerabilities.

3.1. Graph-Based Vulnerability Detection Methods

Graph-based vulnerability detection methods represent programs as structured graphs and learn vulnerability-aware representations from program dependencies. Given a program P , we first transform it into a graph $\mathcal{G}_P$ to capture structural information; each node carries an initial feature vector. A graph neural network then propagates information over the graph to produce node-level, subgraph-level, or graph-level representations for vulnerability detection or localization. To explicitly account for different types of program dependencies, let X denote the node feature matrix and A_r the adjacency matrix corresponding to relation $r \in R$. For graph-level prediction, this process can be formulated as:

$$h_G = \text{READOUT}(\text{GNN}_\theta(X, \{A_r\}_{r \in R})) \quad (1)$$

where $r \in R$ denotes a program relation such as control flow, data flow, call, or program dependence. $\text{GNN}_\theta(\cdot)$ denotes a graph representation learning module that propagates information along these typed program dependencies, and $\text{READOUT}(\cdot)$ aggregates the resulting node representations for downstream prediction.

Table 1 summarizes representative graph-based vulnerability detection methods. Devign [43] fuses AST, CFG, DFG, and natural code sequences into a composite graph, while FUNDED [44] further incorporates control, data, and call dependencies to alleviate the semantic information loss caused by syntax-only representations. However, richer graph structures also increase graph-construction complexity and make vulnerability detection more computationally demanding. ReVeal [45] adopts CPGs to jointly encode syntactic and semantic information and improves robustness on more realistic and imbalanced vulnerability datasets. Nevertheless, these methods spend substantial time on graph construction, which limits their applicability in large-scale data-processing scenarios. In contrast, ReGVD [46] constructs lightweight token co-occurrence graphs to reduce the construction cost and noise introduced by complex program-analysis graphs, but at the expense of explicitly modeling control, data, and call dependencies. These methods have also been extended to Solidity smart contracts: Qian et al. [47] construct a Code

Table 1. Representative graph-based vulnerability detection methods.

Method	Graph Type	Embedding	Architecture	Dataset
Devign [43]	Composite Graph	Word2Vec [48]	GGNN	Devign
FUNDED [44]	extended AST	Word2Vec	Relational GNN	SARD, NVD, GitHub
ReVeal [45]	CPG	Word2Vec	GGNN	ReVeal
IVDetect [49]	PDG sub-graph	GloVe [50]	GGNN with Attention	ReVeal, Devign, BigVul
DeepWukong [51]	eXtended Flow Graph	Doc2Vec [52]	GCN / GAT	SARD, GitHub
BGNN4VD [53]	CPG	Word2Vec	BGNN	NVD, GitHub
LineVD [54]	Line-level CPG	CodeBERT [55]	GAT	BigVul
ReGVD [46]	Token Co-occurrence Graph	GraphCodeBERT [56]	Res-GCN	Devign
CAG [57]	Code Aggregate Graph	Word2Vec / GloVe	GCN / GGNN / GIN	SARD, GitHub
HyVulDect [58]	Semantic CPG	Word2Vec	GGNN	SARD, GitHub
MAGNET [59]	CPG with Meta-paths	Word2Vec	Meta-path Attentional GNN	Devign, ReVeal, BigVul
VulEye [30]	Sub-Dependence Graph	Doc2Vec	GCN	SARD
AMPLE [26]	Simplified Program Graph	Word2Vec	GAT	Devign, ReVeal, BigVul
Graph-BiLSTM [60]	CPG	Sent2Vec [61]	BiLSTM	Juliet, Devign, PrimeVul
EnGS2F [62]	CRG	Doc2Vec	GGNN with Attention	SARD, NVD
MGVD [25]	Multiple graphs	Sent2Vec	GAT	SARD, Devign, BigVul
VulGCANet [63]	CPG	Word2Vec	GCN / GAT	Juliet, DiverseVul
HeVulD [64]	HeVG ^a	Word2Vec	HETConv/HANConv/HGTConv ^b	SARD
DGVD [65]	CFG	CodeBERT	GCN + GAT	Devign
GraphFVD [66]	CPG	GraphCodeBERT	HierGCN	NVD, GITA
HgtJIT [67]	diffPDG	CCT5 [68]	HGT ^c	CodeJIT [69]
GPCVD [70]	CPG	GraphCodeBERT	GGNN	Devign, ReVeal
Qian et al. [47]	Code Semantic Graph	BERT	GAT	Qian et al.
Zhuang et al. [71]	Contract Graph	Statistical Features	DR-GCN / TMP ^d	Ethereum

^a HeVG denotes Heterogeneous Vulnerability Graph. ^b HETConv, HANConv, and HGTConv denote heterogeneous graph convolution.

^c HGT denotes Heterogeneous Graph Transformer. ^d TMP denotes Temporal Message Propagation.

Semantic Graph and evaluate their method on a self-collected dataset, while Zhuang et al. [71] build a Contract Graph, initialize node embeddings with statistical features, and perform temporally ordered message passing to capture execution-related dependencies. However, these studies provide limited investigation of cross-dataset generalization. Together, these works establish the core pipeline of graph-based vulnerability detection: graph construction, node representation, graph neural propagation, and graph-level classification.

Later work further refined graph construction to suppress irrelevant information and support more precise vulnerability detection and localization. IVDetect [49] extracts PDG-based subgraphs and employs attention mechanisms to focus on vulnerability-relevant control and data dependencies, thereby reducing interference from irrelevant program context. DeepWukong [51] constructs vulnerability-oriented eXtended Flow Graphs (XFGs) from program slices to retain relevant control- and data-flow context while filtering out unrelated statements. However, its effectiveness depends heavily on the accuracy of static analysis and program slicing. LineVD [54] performs statement-level vulnerability detection over CPGs. By combining function-level and statement-level predictions, it reduces spurious vulnerable-line candidates and improves the F1-score by approximately 105% on the Big-Vul dataset. However, its reasoning remains largely confined to intra-procedural control and data dependencies within individual functions, limiting its ability to analyze vulnerabilities that require cross-function or cross-file context.

In Table 2, we further compare representative graph-based methods in terms of analysis granularity, cross-procedural context modeling, filtering/suppression mechanisms, and major limitations. Support for cross-procedural context varies markedly among methods. FUNDED accounts for call relations, whereas DeepWukong uses program slicing to widen the vulnerability-relevant context. Explicit cross-procedural modeling, however, remains limited, or even absent, in several representative methods. Filtering and suppression mechanisms extend from dependency-guided context extraction and vulnerability-oriented slicing to joint function- and statement-level prediction.

Space constraints preclude a detailed discussion of other methods, such as HyVulDect [58] and MGVD [25]. While sharing a common representation-learning pipeline, graph-based methods differ mainly in graph construction, node feature computation, graph learning mechanisms, and detection granularity. Most existing methods, however, rely heavily on third-party program analysis tools—Joern being a prominent example—for graph construction. They routinely discard samples that cannot be parsed or whose generated graphs exceed a predefined node-size threshold [72]. Such filtering can introduce evaluation bias: reported performance may be driven largely by successfully parsed, structurally regular samples rather than the full diversity of real-world vulnerable code. Robustness to parsing failures and scalability to large graphs have not yet been systematically addressed, which leaves an important gap. Improving the robustness, scalability, and fault tolerance of program graph construction therefore remains a pressing direction for future research.

Table 2. Comparison of representative graph-based vulnerability detection methods.

Method	Granularity	Cross-Procedural Context	Filtering/Suppression	Main Limitation
VulEye [30]	Subgraph/Slice	Limited; no explicit interprocedural modeling	Sensitive-function-centered PDG slicing	Depends on sensitive-function-based slicing
FUNDED [44]	Function	Partial; call relations included	No explicit filtering	Requires construction of multiple dependency relations
ReGVD [46]	Function	No; no explicit cross-procedural modeling	No explicit filtering	Lacks explicit control/data/call semantics
IVDetect [49]	Function / Subgraph	Limited; mainly function-level	Dependency-guided context extraction	Limited cross-function context
DeepWukong [51]	Program Slice	Not explicit; slice-based context	Vulnerability-oriented slicing	Sensitive to slicing quality
LineVD [54]	Function/Statement	Limited; mainly intra-procedural	Joint function/statement prediction	Limited cross-function/cross-file context

3.2. Graph-Based Multimodal Vulnerability Detection Methods

Graph-based multimodal vulnerability detection methods reduce information loss by integrating heterogeneous representations of the same program. A typical framework consists of three components: a modality constructor that extracts multiple information sources (e.g., code sequences, ASTs, graph structures, and LLM-guided contexts); a modality encoder that learns modality-specific representations using suitable neural architectures; and a multimodal fusion module that aligns and integrates these heterogeneous features for vulnerability detection or localization. Let $\mathcal{M} = \{m_G, m_1, \dots, m_{k-1}\}$ denote the set of constructed modalities, where m_G is the graph modality. To explicitly preserve program-specific structural constraints, the graph modality is encoded as $z_G = \text{ENC}_G(X, \{A_r\}_{r \in R})$, where A_r denotes the relation-specific adjacency matrix for relation r , such as syntactic, control-flow, data-flow, call, and program-dependence relations. The remaining modalities are encoded as $z_i = \text{ENC}_i(m_i)$. The multimodal prediction process can then be formulated as

$$\hat{y} = f_\theta(\text{FUSE}(z_G, z_1, \dots, z_{k-1})), \quad (2)$$

where $\text{FUSE}(\cdot)$ denotes a generic fusion operator, such as concatenation, attention-based alignment, contrastive alignment, or knowledge distillation, and f_θ is the downstream prediction head.

Table 3 summarizes representative graph-based multimodal vulnerability detection methods. MGVD [25] combines AST, PDG, and code-sequence information through multiple graph-based channels to mitigate information loss from a single representation, improving the F1-score by 9.68–10.28%. VulMPFF [73] integrates sequential, lexical/syntactic, and graph-structural features, although direct feature concatenation may introduce redundancy. GRACE [74] augments LLM inputs with graph structure and retrieved demonstrations, addressing the structural limitations of plain-text prompts and achieving an F1-score improvement of at least 28.65% on the evaluated benchmarks. DeepFusion [75] combines AST representations with program slices to jointly capture structural and semantic information. LLMxCPG [76] further employs CPG-guided slicing to retain vulnerability-relevant context while removing irrelevant graph regions. This reduces code size by 67.84–90.93%, supports analysis across multiple functions, and improves F1-score by 15–40% over the compared baselines, although its effectiveness depends on reliable CPG construction and query generation. MCL-VD [77] uses multimodal contrastive learning to align textual and structural representations, while Vul-LMGNNs [78] transfers knowledge between language-model and graph-based representations through online distillation. More recent methods, including MVulD+ [72], Cai et al. [79], and CMF-Vul [80], extend this direction by jointly modeling graph, textual, and image-like representations.

In summary, these methods show that combining heterogeneous program views can improve semantic coverage, suppress irrelevant information, and broaden the code context available for vulnerability reasoning. However, multimodal integration also introduces additional computational overhead and may propagate redundant or noisy features across modalities. Thus, the main challenge is to balance representation completeness with efficient and reliable modality alignment.

3.3. LLM-Assisted Vulnerability Detection

LLM-assisted vulnerability detection methods primarily use large language models as the main reasoning component. These methods include retrieval-augmented generation (RAG), prompt optimization, fine-tuning,

Table 3. Representative graph-based multimodal vulnerability detection methods.

Method	Graph Modality	Other Modalities	Fusion Strategy	Datasets
MGVD [25]	AST + PDG	Text information	Three-channel feature generation	SARD, Devign, BigVul
VulMPFF [73]	CPG	Text information + Syntactic relations	Multi-perspective feature concatenation	Devign, ReVeal, BigVul
GRACE [74]	CPG	LLM	Structured prompts for LLMs	Devign, ReVeal, BigVul
DeepFusion [75]	AST	Program slice information	Converts ASTs and code slices into unified word representations	Manually collected dataset
LLMxCPG [76]	CPG	LLM	LLM for optimizing CPG query language	SVEN, ReposVul
CodeSAGE [81]	Code Attribute Graph	Text information + Statistical features	Attention-based alignment	SARD, Devign
Vul-LMGNNs [78]	CPG	Text information	Online knowledge distillation	DiverseVul, Devign, ReVeal, Draper VDSIC
MCL-VD [77]	AST	Text information	Multi-modal contrastive learning	Devign, ReVeal, BigVul
CPGHunter [82]	CPG	LLM	Semantic rule generation for CPG paths	Juliet, CWE-Bench-Java
MVulD+ [72]	CPG	Text information + Image information	Align and fuse the code representations	BigVul
Cai et al. [79]	CPG	Text information + Image information	Conversion of CPGs into image matrices and use of ViLT ^a for detection	SARD, GitHub
CMF-Vul [80]	PDG	Text information + Image information	Mapping different representations into a shared latent space	SARD, BigVul

^a ViLT denotes Vision-Language Transformer.

multi-agent collaboration, and tool-assisted analysis. LLM-assisted methods span code- and knowledge-based reasoning as well as approaches that incorporate program graphs, graph-derived code slices, GNN representations, or static-analysis evidence. Graph-LLM hybrid methods such as GRACE, LLMxCPG, and CPGHunter are covered in Section 3.2 as graph-based multimodal approaches. The overall process can be formulated as:

$$\hat{o} = f\left(\text{Reason}\left(\text{LLM}_\theta(P, \mathcal{G}_P, \mathcal{K})\right)\right). \quad (3)$$

where P denotes the target program, $\mathcal{G}_P$ represents structural program information when available, including ASTs, CFGs, PDGs, CPGs, graph-derived code slices, or GNN-learned structural representations, and $\mathcal{K}$ denotes auxiliary security evidence, such as retrieved vulnerability descriptions, historical examples, CWE/NVD information, or static-analysis results. $\text{Reason}(\cdot)$ abstracts the reasoning process over program semantics, structural information, and external knowledge, with optional validation or refinement based on self-validation, external evidence, or static-analysis constraints. The output mapping f converts the reasoning result into the final vulnerability detection output $\hat{o}$ through either a learned prediction head or a parameter-free output parser.

Table 4 summarizes representative LLM-assisted vulnerability detection methods according to their primary reasoning and adaptation strategies. Retrieval-augmented generation (RAG) mainly addresses the limited vulnerability-specific knowledge of general-purpose LLMs by introducing external security evidence into the reasoning process. VulInstruct [83], Vul-RAG [84], ProveRAG [85], and VulScribeR [86] retrieve vulnerability descriptions, historical cases, or security specifications and incorporate them into prompts. In particular, Vul-RAG [84] organizes vulnerability knowledge at the knowledge level and reports substantial improvements in balanced precision and recall over direct LLM-based detection under its benchmark setting, demonstrating the benefit of incorporating vulnerability-specific knowledge. ProveRAG [85] further introduces provenance-aware retrieval and self-critique to reduce unsupported conclusions, while RAG-SmartVuln [87] combines fine-tuning with retrieval augmentation for smart-contract vulnerability detection. However, these methods remain sensitive to retrieval quality and knowledge-base coverage; irrelevant or incomplete evidence may instead introduce additional reasoning noise.

Multi-agent collaboration addresses another limitation of single-LLM detection: vulnerability judgments may be affected by hallucination or one-sided reasoning. AutoReview [88] allows multiple agents to cooperate over security-oriented code contexts, while VulTrial [89] introduces an adversarial courtroom-style process in which different agents argue for and against vulnerability findings before a final judgment. Agent4Vul [90] further combines agent-based reasoning with GNN-derived features to integrate structural and semantic evidence. Under

Table 4. Representative LLM-assisted vulnerability detection methods.

Method	Base Model	Architecture	Reasoning Strategy	Datasets
VulInstruct [83]	GPT-OSS-120B/Claude Sonnet 4/DeepSeek-R1	LLM + RAG	RAG-based prompting	PrimeVul
ProveRAG [85]	GPT-3.5-Turbo	LLM + RAG	RAG-based prompting with a self-critique mechanism	NVD
Vul-RAG [84]	GPT-4o/Claude 3.5/DeepSeek-Coder/Qwen	LLM + RAG	RAG-based prompting	PairVul
VulScribeR [86]	GPT-3.5/CodeQwen1.5	LLM + RAG	RAG-based prompting	Devign, ReVeal, BigVul, PrimeVul
RAG-SmartVuln [87]	Qwen2.5-Coder/GPT-3.5-Turbo	LLM + RAG	LLM fine-tuning with RAG	SolidiFI [91], SmartBugs
AutoReview [88]	DeepSeek-Coder	Multi-Agent	CPG-based RAG with graph-guided slicing for repair prompting	ReposVul
VulTrial [89]	GPT-3.5/GPT-4o/Llama 3.1	Multi-Agent	Courtroom-inspired adversarial reasoning	PrimeVul
Agent4Vul [90]	GLM / Qwen2.5	Multi-Agent	Agent Chain-of-Thought (CoT) prompting with GNN feature fusion	Qian et al.
GPTLens [92]	GPT-4 / GPT-3.5-Turbo	Multi-Agent	Auditor–critic vulnerability generation and discrimination	SmartBugs
LLMVulExp [36]	Llama 3 / Code Llama	Single Model	Instruction tuning with Low-Rank Adaptation (LoRA) and CoT-based key-code extraction	SeVC, DiverseVul, PrimeVul
SecureForge [93]	Qwen2.5/GPT-5/Claude Sonnet 4.6/DeepSeek-V3/Code Llama/Kimi-K2	Single Model	Prompt optimization with Markov chain Monte Carlo sampling	SWE-Chat [94]
VULSOLVER [95]	DeepSeek-V3/Kimi-K2/GPT-4o/GPT-4-Turbo	LLM + Static Analysis	Static-analysis-guided LLM constraint solving	OWASP [96], SeVC
GPTScan [97]	GPT-3.5-Turbo	LLM + Static Analysis	GPT-based semantic understanding with static-analysis validation	Top200 [98], Web3Bugs [99]

the benchmark comparisons reported in the original study, Agent4Vul achieves a 3.61% higher F1-score than the compared baselines, indicating that multimodal structural information and collaborative reasoning can complement each other. GPTLens [92] similarly employs an auditor–critic architecture, where the critic filters and refines candidate vulnerabilities generated by the auditor. Nevertheless, multi-agent systems require substantially more inference calls and their effectiveness depends on whether the participating agents provide genuinely complementary reasoning rather than propagating the same erroneous evidence.

Other methods focus on model adaptation, prompt optimization, or tool-guided reasoning to improve vulnerability-specific reasoning. LLMVulExp [36] applies instruction tuning and LoRA together with CoT-based key-code extraction to focus the model on vulnerability-prone regions, while SecureForge [93] optimizes vulnerability-related prompts to improve detection behavior. VULSOLVER [95] formulates vulnerability detection as an LLM-guided constraint-solving problem, addressing the difficulty of reasoning about path-sensitive vulnerability conditions. GPTScan [97] instead combines GPT-based semantic reasoning with program-analysis validation, using static analysis to verify candidate logic vulnerabilities and suppress unsupported predictions. However, tool-guided methods inherit the limitations of the underlying static analysis: incomplete path exploration or inaccurate program-analysis results would prevent the LLM from receiving the evidence required to identify a vulnerability.

Overall, these studies show that LLM-assisted vulnerability detection is evolving from direct code prompting toward evidence-grounded reasoning. RAG methods address missing security knowledge, multi-agent approaches mitigate unreliable single-model judgments, and graph- or static-analysis-assisted methods provide structural evidence for more constrained reasoning.

3.4. Vulnerability Repair

Building on the repair task formulation in Section 2, this subsection focuses on graph-driven methods for generating and validating patches. The goal is not only to identify vulnerable code regions, but also to synthesize concrete code transformations that eliminate vulnerability-triggering dependencies while preserving intended functionality.

A typical evidence-grounded repair framework consists of four modules. The evidence constructor gathers the vulnerable code, optional structural representations (ASTs, CFGs, PDGs, CPGs, code slices, edit scripts, data-flow facts), static-analysis reports, retrieved fixes, prompts, or runtime feedback. The localization and reasoning module identifies the repair region and explains the vulnerability. The patch-generation module produces candidate edits via templates, mined repair patterns, neural generation, LLM prompting, or retrieval-augmented synthesis. The validation module ranks candidates and checks whether the vulnerability has been removed while preserving semantics.

The methods in Table 5 represent the major repair directions. AST- and edit-evidence-based methods exploit structural similarity between vulnerable code and historical fixes. SimFix [100] searches for similar donor code and adapts patches through AST differencing; TBar [101] matches suspicious AST contexts with predefined repair templates; FixMiner [102] mines reusable repair patterns from rich edit scripts; and VulRep [103] combines AST abstraction with fixing-commit evidence for vulnerability repair. These approaches effectively constrain the repair space, but their performance depends strongly on the availability and coverage of donor code, templates, or historical repair patterns. However, AST-centered representations mainly capture syntactic relations and may provide insufficient information for vulnerabilities whose causes depend on long-range data flow, control dependencies, or cross-procedural interactions.

Methods that treat program structure as a direct repair signal form a second direction. Hoppity [104] formulates repair as graph-edit prediction over AST-derived program graphs; Recoder [105] uses AST subtree localization and syntax-guided neural edit decoding; and SPVR [106] converts AST-difference evidence and CWE labels into structured prompts for LLM-based vulnerability repair. These methods are closely related to graph-based repair because syntax and graph evidence directly constrain the edit space and guide concrete program transformations.

Neural patch-generation methods learn mappings from vulnerable code contexts to fixed code. SeqTrans [107] augments Transformer-based repair with data-flow evidence and AST-diff pairs; SequenceR [108] uses abstract buggy contexts and a copy mechanism to generate fixed lines; CoCoNuT [109] separates buggy-line and surrounding-context encoders for context-aware repair; VRepair [110] transfers generic bug-fixing knowledge to vulnerability repair; VulRepair [111] adopts CodeT5-style subword modeling for vulnerability-specific patch generation. On 8482 vulnerability fixes from 1,754 real-world projects, VulRepair achieves a 44% Perfect Prediction, which is 13–21% higher than the compared baselines, and correctly repairs 745 of 1706 real-world vulnerabilities; and VulMaster [112] fuses segmented code, AST evidence, and CWE-oriented examples.

LLM- and Pre-trained Language Model (PLM)-based repair methods shift the interface from internal representations to prompts and feedback. ContractTinker [113] uses dependency-graph evidence and program slices to guide smart-contract patching; Xia et al. [114] evaluate PLM-based repair through generation, infilling, single-line repair, and patch ranking; ChatRepair [115] introduces conversational repair with failed-test feedback and previous patch attempts. These works show that prompts can carry structural, semantic, and validation evidence even without direct graph learning. However, these results concern general program repair rather than vulnerability-specific benchmarks, and iterative LLM interaction introduces additional inference cost.

RAG-, static-analysis-, and agent-based repair methods incorporate external evidence and iterative validation into patch synthesis. RAP-Gen [116] retrieves bug-fix patterns before CodeT5-based generation; InferFix [117] combines Infer static-analysis reports, retrieval augmentation, prompt augmentation, and Codex fine-tuning; VulDebugger [118] uses an LLM agent to compare expected and actual runtime states during repair. These methods indicate a shift from isolated patch generation toward evidence-grounded and validation-aware repair. These approaches improve evidence grounding and reduce unconstrained generation, but their effectiveness depends on retrieval quality, static-analysis accuracy, and the availability of informative runtime or testing feedback.

Collectively, vulnerability repair extends graph-driven analysis from identifying vulnerable program elements to generating concrete transformations that break vulnerability-triggering dependencies. By combining structural evidence, historical edit patterns, neural generation, retrieval context, static-analysis facts, and LLM feedback, recent methods improve the controllability and verifiability of patch generation.

4. Libraries for Vulnerability Detection

This section reviews the datasets, program graph construction tools, and computational frameworks that support graph-driven vulnerability detection. We first summarize representative datasets and their annotation characteristics, then discuss tools for constructing program graphs, and finally introduce graph learning, Transformer, LLM serving, and RAG frameworks used to build scalable vulnerability analysis pipelines.

Table 5. Representative vulnerability repair methods.

Method	Core Repair Strategy	Input / Evidence Type
SimFix [100]	Similar-code repair using patch-space intersection, AST differencing, and donor-code adaptation	AST/edit evidence, donor-code context, usage sequences
TBar [101]	Template-based repair using AST-context matching and predefined fix patterns	AST context, repair templates, suspicious statements
FixMiner [102]	Fix-pattern mining using iterative clustering of rich edit scripts	AST/edit evidence, code-context trees
VulRep [103]	Transformer-based vulnerability repair using AST abstraction and fixing-commit rules	AST abstraction, token-sequence context, fixing-commit evidence
Hoppity [104]	Graph-based repair using GNN-encoded AST transformations and LSTM edit control	Graph context, AST structure, token/value edges
Recoder [105]	Syntax-guided repair using AST subtree localization and neural edit decoding	AST context, edit sequences, placeholders
SPVR [106]	LLM-based vulnerability repair using AST-difference-derived MET prompts and CWE guidance	Prompt context, AST/MET evidence, CWE labels
SeqTrans [107]	Data-flow-guided vulnerability repair using seq2seq Transformer generation and vulnerability-specific fine-tuning	Token-sequence context, data-flow evidence, AST-diff pairs
SequenceR [108]	Seq2seq program repair using abstract buggy-context encoding and copy-based fixed-line generation	Token-sequence context, abstract buggy context
CoCoNuT [109]	Context-aware neural repair using separate buggy-line/context encoders and ensemble-based fixed-line generation	Token-sequence context, buggy-line and surrounding-code context
VRRepair [110]	Transformer-based vulnerability repair using transfer learning and vulnerability-specific fine-tuning	Token-sequence context, CWE/location markers, edit-script evidence
VulRepair [111]	CodeT5-based vulnerability repair using subword tokenization and vulnerability-specific fine-tuning	Subword token-sequence context, location markers
VulMaster [112]	CodeT5-based vulnerability repair using FiD fusion of code, AST, and ChatGPT-augmented CWE examples	Token-segment context, AST evidence, CWE example prompts
ContractTinker [113]	LLM-based smart-contract repair using CDG-guided CoT localization, mitigation reasoning, and patch refinement	Prompt context, dependency-graph evidence, program slices
Xia et al. [114]	Direct PLM repair using function generation, code infilling, single-line generation, and entropy-based patch ranking	Token-sequence context, infill/prefix-suffix prompts
ChatRepair [115]	Conversational ChatGPT repair using test-failure feedback and prior patch attempts	Prompt context, test-failure evidence, patch-feedback history
RAP-Gen [116]	Retrieval-augmented CodeT5 repair using BM25/DPR fix-pattern retrieval and bug-fix-guided generation	Retrieval evidence, token-sequence context, error context
InferFix [117]	Static-analysis-guided LLM repair using retrieval augmentation, prompt augmentation, and Codex fine-tuning	Static-analysis evidence, prompt context, retrieval evidence
VulDebugger [118]	LLM-agent vulnerability repair using dynamic debugging, expected-state reasoning, and runtime validation	Dynamic-runtime evidence, crash constraints, static-code context

4.1. Datasets

Code vulnerability samples constitute the foundational data for vulnerability detection research. These datasets typically contain source code snippets at the function or file level, along with associated labels: binary vulnerability indicators, vulnerability types (e.g., CWE identifiers), and vulnerability location annotations. They span multiple mainstream programming languages: C/C++, Python, Go, et al. and support research stages from traditional static analysis to deep learning-based detection. To facilitate use, we categorize these datasets by data source (synthetic vs. real-world) and labeling method, provide their sample sizes, and list them in chronological order of publication. Given the large number and diversity of datasets, the table also records the original literature source for each entry.

Table 6 summarizes several widely used vulnerability datasets. NIST’s Software Assurance Reference Dataset (SARD) and its derived Juliet Test Suite [119] are the most common synthetic benchmarks. They inject over a hundred CWE types, yet the samples tend to be less challenging than real-world code. Real-world datasets paint a different picture. BigVul [120], CVEFixes [121], and ReposVul [122], all mined from production Git repositories, provide massive function-level vulnerability samples with patch information; they have been widely adopted for deep learning-based vulnerability detection and localization. A key caveat: these version-control-based datasets locate vulnerable regions by analyzing fix-commit diffs. Their limitation is that they can only parse deleted or

Table 6. Representative vulnerability detection datasets.

Dataset	Year	Source	Language	CWE	Line-Level Labels	Vulnerable	Total	Ratio *
Juliet [119]	2017	NIST SARD	•	118	Yes	64,099	N/A	N/A
Draper VDSIC [125]	2018	Juliet, Debian, GitHub	•	N/A	No	87,804	1,286,262	6.8%
VulDeePecker [126]	2018	NVD + SARD	•	2	No	43,119	181,641	23.7%
Deign [43]	2019	Qemu + FFmpeg	•	N/A	No	12,460	27,318	45.6%
BigVul [120]	2020	GitHub + NVD	•	91	Yes	10,900	188,636	5.8%
SmartBugs [127]	2020	Ethereum, GitHub	◇	10	Yes	143	N/A	N/A
SeVC [128]	2021	NVD+SARD	•	126	No	14,780	15,591	94.8%
CrossVul [123]	2021	GitHub + NVD	• ▲ ◆ ◀ ▶ ★ ▼	168	Yes	13,738	27,476	50.0%
CVEFixes [121]	2021	NVD	• ▲ ◆ ◀ ▶ ★ ▼ □	180	Yes	23,897	50,322	47.5%
VulDeeLocator [129]	2022	NVD + SARD	•	N/A	Yes	40,450	157,692	28.8%
ReVeal [45]	2022	Chromium + Debian	•	N/A	No	2240	22,734	9.9%
DiverseVul [130]	2023	GitHub	•	150	No	18,945	330,492	5.7%
Qian et al. [70]	2023	Ethereum, GitHub	◇	4	No	7721	514,880	1.5%
MegaVul [131]	2024	GitHub + NVD	•	169	No	17,380	339,548	5.1%
PrimeVul [132]	2024	GitHub + NVD	•	140	Yes	6968	235,768	3.0%
ReposVul [122]	2024	Mend, GitHub	• ◆	236	Yes	6897	262,216	2.6%
CleanVul [124]	2024	GitHub	◆ ▲	10	No	8198	17,224	47.6%
ICVul [133]	2025	GitHub	•	146	No	6276	15,396	41.0%
GITA [66]	2025	GitHub	•	56	No	10,666	32,466	32.9%
CWE-Bench-Java [134]	2025	GitHub	■	4	No	120	N/A	N/A

Notations. * Ratio denotes Vulnerable/Total (%); N/A indicates insufficient data for calculation. • C/C++, ▲ C#, ◆ Python, ■ Java, ▶ JavaScript, ◀ Go, ★ PHP, ▼ Ruby, □ TypeScript, ◇ Solidity.

modified lines; for vulnerabilities introduced by newly added code, they often fail to pinpoint the original vulnerable region, which compromises annotation accuracy to some degree. Cross-lingual evaluation often relies on datasets like CrossVul [123], CVEFixes, and CleanVul [124], which span multiple languages. Because some datasets bundle vulnerable and non-vulnerable code, patch diffs, and CWE labels, they enable not only detection but also finer-grained compound tasks: localization and classification. Due to space constraints, more datasets will not be analyzed or discussed in detail.

4.2. Program Graph Construction Tools

Program graph construction tools form the bedrock of graph-driven vulnerability intelligence research. These tools automatically parse source code, extract multi-dimensional syntactic and semantic dependencies, and generate a variety of standardized graph representations: Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graphs (DFG), Program Dependence Graphs (PDG), Code Property Graphs (CPG), call graphs, System Dependence Graphs (SDG), and Static Single Assignment (SSA)-based intermediate representations. The completeness and granularity of these graphs set an upper bound on what vulnerability reasoning models can achieve. Table 7 summarizes nine widely used tools, along with their key features and language coverage.

Joern [135] is among the most widely used integrated CPG construction tools. It generates AST, CFG, PDG, DFG, and CPG within a unified analysis pipeline, supports over ten languages (C/C++, Java, Python, among others), and preserves fine-grained node attributes such as token content, operator type, and API information. Joern has become a standard tool for graph-driven vulnerability detection, although parsing speed remains a concern on ultra-large-scale projects. CodeQL [136] brings a different design: rich graph structures, including AST, CFG, call graphs, and data-flow graphs, spanning over ten languages (C/C++, Go, Java, JavaScript, Python, Rust, TypeScript). Its tight integration with a declarative query language turns program graphs into a queryable relational model, granting unusual flexibility for custom vulnerability pattern matching; it is widely employed in variant analysis and security auditing. Tree-sitter [137] and ANTLR [138] focus primarily on efficient AST construction. Tree-sitter handles real-time, incremental parsing across dozens of languages, making it ideal for lightweight syntax modeling, whereas ANTLR provides grammar-driven parsing for multiple mainstream languages and serves as a flexible front-end for custom program analysis. For C/C++ (and Rust, via LLVM), LLVM [139] and Frama-C [140] deliver high-precision CFGs and intermediate representations; Frama-C additionally produces PDGs, making both tools suitable for rigorous semantic analysis. WALA [141] and Soot [142] concentrate on the Java ecosystem, with WALA also supporting JavaScript, Python, and Solidity, and generate CFGs, call graphs, and system dependence graphs; they are widely applied in program slicing and static analysis. Slither [143] is purpose-built for Solidity smart contracts, generating ASTs, CFGs, and call graphs, which makes it particularly valuable for blockchain

Table 7. Tools for generating code graphs.

Tool	Generated Graphs	Support Languages
Joern [135]	AST, CFG, PDG, DFG, CPG	●■◆▶◀★◆▶◀
CodeQL [136]	AST, CFG, Call Graph, Data Flow Graph	●▲◀■▶◆◆▶◀■
Tree-sitter [137]	AST	●■◆▶◀★◆▶◀■
WALA [141]	CFG, Call Graph, SDG	■▶◆▶
Soot [142]	CFG, Call Graph	■●
ANTLR [138]	AST	●▲■▶■◆▶◀★◆▶◀
Slither [143]	AST,CFG,Call Graph	▶
LLVM [139]	CFG, SSA IR Graph	●▲
Frama-C [140]	AST, CFG, PDG	●

Notations. ● C/C++, ▲ C#, ◆ Python, ■ Java, ▶ JavaScript, ◀ Go, ★ PHP, ▼ Ruby, ■ TypeScript, ◆ Kotlin, ▲ Rust, ◀ Swift, ● Android, ▶ Solidity.

vulnerability analysis.

In practice, integrated multi-graph tools such as Joern and CodeQL capture complex vulnerability semantics well; lightweight parsers like Tree-sitter and ANTLR excel when speed and syntax-level modeling are paramount. For smart-contract security, Slither fills a specific niche. Meanwhile, LLVM, Frama-C, WALA, and Soot remain essential for constructing high-precision control-flow, data-flow, dependence, and call graphs in compiler-centric or static-analysis workflows.

4.3. Computing Frameworks

The computational stack for graph- and LLM-based vulnerability detection comprises four complementary layers: graph learning frameworks, code representation and Transformer frameworks, LLM inference and serving frameworks, and retrieval-augmented generation frameworks.

4.3.1. Graph Learning Frameworks

Table 8 summarizes representative graph learning computational frameworks and their main roles in vulnerability detection. At the foundation, PyTorch [144] provides automatic differentiation and GPU acceleration. For graph-structured program representations such as ASTs, CFGs, and CPGs, PyTorch Geometric (PyG) [145] integrates tightly with PyTorch and offers flexible data objects and a rich suite of GNN operators; it is consequently a popular choice for node-, edge-, and graph-level vulnerability classification. Deep Graph Library (DGL) [146] follows a graph-centric paradigm, excelling at heterogeneous message passing and sampling-based training; this makes it particularly well-suited for multi-relation code property graphs that weave together syntactic, control, data, and call edges. NetworkX [147] serves as a preprocessing tool: cleaning parser outputs, pruning vulnerability-related program slices, aligning multiple program views, and converting graph structures into the tensor formats expected by graph learning frameworks. Frameworks built on TensorFlow and PaddlePaddle, such as Tf-geometric [148] and Paddle Graph Learning (PGL) [149], offer alternative GNN backends; PGL additionally supports distributed graph learning. CogDL [150] and Dive into Graphs (DIG) [151] complement the ecosystem through unified model libraries, benchmark suites, and research-oriented graph learning tasks. For large-scale or system-level graph computation, AliGraph [152], Euler [153], Roc [154], GraphScope [155], GeaFlow [156], cuGraph [157], and GraphVite [158] provide distributed training, graph embedding, streaming processing, GPU-accelerated analytics, and high-throughput graph computation. Collectively, these frameworks span the pipeline of graph-based vulnerability detection, from program graph preprocessing and transformation, through scalable GNN training and graph embedding, to distributed graph processing and analytics. A more exhaustive listing is omitted here; space constraints prevent a deeper dive.

4.3.2. Transformer Frameworks

Transformer-based vulnerability detection leans heavily on general-purpose sequence modeling frameworks and large-model training infrastructures. Table 9 summarizes representative Transformer-related computational frameworks and their main roles in vulnerability detection. Hugging Face Transformers [159] provides a unified interface for loading, fine-tuning, and evaluating pre-trained language and code models; it is consequently a widely adopted platform for adapting BERT-like encoders, encoder-decoder models, and decoder-only LLMs to tasks such as vulnerability classification, ranking, and repair. PEFT enables parameter-efficient adaptation through

Table 8. Graph learning computational frameworks for vulnerability detection.

Framework	GitHub Link	Main Role and Strength
PyTorch [144]	pytorch/pytorch	Deep learning backend with a mature ecosystem.
PyTorch Geometric (PyG) [145]	pyg-team/pytorch_geometric	Flexible GNN modeling tightly integrated with PyTorch.
Deep Graph Library (DGL) [146]	dmlc/dgl	Scalable GNN training with heterogeneous graph support.
NetworkX [147]	networkx/networkx	General-purpose Python graph analysis and preprocessing library.
Tf-geometric [148]	CrawlScript/tf_geometric	TensorFlow-based GNN toolkit with lightweight APIs.
Paddle Graph Learning (PGL) [149]	PaddlePaddle/PGL	PaddlePaddle-based graph learning with distributed support.
CogDL [150]	THUDM/CogDL	Unified graph learning toolkit with broad model coverage.
AliGraph [152]	alibaba/graph-learn	Distributed framework for large-scale industrial GNN training and inference.
Euler [153]	alibaba/euler	Distributed graph embedding and GNN training at scale.
Roc [154]	jiazhihao/Roc	Multi-GPU GNN execution with high training throughput.
GraphScope [155]	alibaba/GraphScope	Unified graph computing across analytics, traversal, and learning.
GeaFlow [156]	apache/geaflow	Streaming graph computation for dynamic graph processing.
cuGraph [157]	rapidsai/cugraph	GPU-accelerated graph analytics and scalable graph algorithms.
Dive into Graphs (DIG) [151]	dive-lab/DIG	Research-oriented graph learning with advanced task support.
GraphVite [158]	DeepGraphLearning/graphvite	High-performance graph embedding with CPU–GPU hybrid training.

lightweight techniques, including LoRA and prompt tuning, while AdapterHub [160] supplies modular Transformer adapters and reusable task-specific components. Accelerate [161] supports mixed-precision and distributed training, simplifying deployment across diverse hardware configurations. fairseq [162] and OpenNMT-py [163] offer modular sequence-modeling pipelines and reproducible training recipes; these prove especially useful when vulnerability detection is cast as sequence classification, sequence labeling, or sequence-to-sequence learning.

Earlier systems Tensor2Tensor [164], Marian [165], Sockeye [166] deliver mature implementations of Transformer-based sequence modeling and neural machine translation. They round out the ecosystem for code summarization, code translation, and vulnerability-related program transformation. For large-scale code model training, Megatron-LM [167], DeepSpeed [168], and Colossal-AI [169] target model parallelism, memory optimization, and distributed training, which makes them well-suited for scaling Transformer architectures to massive source-code corpora. Additional engineering tools NVIDIA Transformer Engine, torchtune, and LitGPT further support low-precision acceleration, reproducible fine-tuning, and lightweight LLM development. Collectively, these frameworks span the Transformer-based vulnerability detection pipeline, from tokenization and pre-trained model adaptation through scalable training and efficient fine-tuning to deployment. A more exhaustive listing is omitted; space constraints do not permit a deeper dive.

4.3.3. LLM Inference Frameworks

LLM-based vulnerability analysis rests on inference and serving frameworks that govern how large language models are executed, deployed, and accessed in practical security workflows. Table 10 summarizes representative

Table 9. Transformer-related computational frameworks for vulnerability detection.

Framework	GitHub Link	Main Role and Strength
Hugging Face Transformers [159]	huggingface/transformers	Pretrained model framework for loading, training, and inference.
PEFT ^a	huggingface/peft	Enables parameter-efficient adaptation through LoRA, adapters, and prompt tuning.
Accelerate [161] ^a	huggingface/accelerate	Simplifies mixed-precision and distributed training across hardware settings.
AdapterHub [160]	adapter-hub/adapters	Supports modular Transformer adapters and reusable task-specific adaptations.
fairseq [162]	facebookresearch/fairseq	Sequence-modeling toolkit with Transformer reference implementations.
OpenNMT-py [163]	OpenNMT/OpenNMT-py	PyTorch NMT framework for translation and NLP tasks.
Tensor2Tensor [164]	tensorflow/tensor2tensor	TensorFlow toolkit with early Transformer training pipelines.
Marian [165]	marian-nmt/marian	Efficient C++ NMT framework with Transformer training and inference.
Sockeye [166]	aws-labs/sockeye	Maintenance-mode PyTorch NMT toolkit with distributed training and optimized inference.
Megatron-LM [167]	NVIDIA/Megatron-LM	Enables large-scale Transformer pre-training through tensor, pipeline, and data parallelism.
NVIDIA Transformer Engine ^a	NVIDIA/TransformerEngine	Accelerates Transformer training and inference with optimized low-precision kernels.
DeepSpeed [168]	microsoft/DeepSpeed	Provides ZeRO optimization and model sharding for large-scale model training.
Colossal-AI [169]	hpcaitech/ColossalAI	Supports distributed Transformer training, parallelism, and memory optimization.
torch tune ^a	pytorch/torch-tune	Provides PyTorch-native recipes for fine-tuning, evaluating, and post-training LLMs.
LitGPT ^a	Lightning-AI/litgpt	Offers readable recipes for pre-training, fine-tuning, evaluating, and deploying LLMs.

^a Auxiliary engineering tools for training, adaptation, and acceleration.

LLM inference and serving frameworks and indicates their roles in vulnerability analysis. vLLM [170] delivers high-throughput serving by leveraging efficient attention and batching (a good fit for large-scale scanning where many parallel code queries are needed). SGLang [171] handles structured, multimodal, and agentic LLM workloads efficiently, proving valuable when vulnerability analysis involves multi-step reasoning, tool use, or structured outputs. DeepSpeed-MII [172] extends high-performance serving via DeepSpeed inference optimizations, while Sarathi-Serve [173] is a research-oriented system designed to improve the throughput–latency trade-off. For local, private analysis, llama.cpp and Ollama provide lightweight execution and management of open-weight models on controlled hardware. WebLLM [174] pushes local execution one step further, into the browser, with WebGPU-accelerated inference. For GPU-accelerated, production-oriented deployment, LMDeploy, TensorRT-LLM, and NVIDIA Triton Inference Server offer quantization, optimized kernels, batching, multi-backend serving, and distributed deployment. MLC-LLM complements these tools by compiling and deploying LLMs across heterogeneous platforms through machine learning compilation. FastChat [175] delivers chatbot training, evaluation, and multi-model serving, serving as a convenient platform for prototyping LLM-based vulnerability assistants. Ray, through Ray Serve, adds a distributed runtime and scalable serving layer, supporting larger pipelines that demand batching, parallelism, and service composition. In practice, these frameworks cover the full spectrum: local model execution and browser-side inference, high-throughput serving, GPU acceleration, distributed deployment, and production orchestration. Due to space limitations, more frameworks and tools are not discussed in detail.

4.3.4. Retrieval-Augmented Generation (RAG) Frameworks

RAG-based vulnerability intelligence leans on retrieval, indexing, orchestration, and evaluation frameworks that connect LLMs with external security knowledge, code repositories, vulnerability reports, and historical patches. Table 11 summarizes representative RAG-related frameworks and systems and their main roles in vulnerability

Table 10. LLM inference and serving frameworks for vulnerability detection and analysis.

Framework	itHub Link	Main Role and Strength
vLLM [170]	vllm-project/vllm	High-throughput LLM serving with efficient attention and batching.
llama.cpp	ggml-org/llama.cpp	Lightweight quantized LLM inference across CPUs and GPUs.
Ollama	ollama/ollama	Local execution and management of open-weight language models.
SGLang [171]	sgl-project/sglang	Efficient serving for structured, multimodal, and agentic LLM workloads.
LMDeploy	InternLM/lmdeploy	Efficient LLM deployment with quantization and distributed serving.
TensorRT-LLM	NVIDIA/TensorRT-LLM	NVIDIA-optimized LLM inference with specialized GPU kernels.
NVIDIA Triton Inference Server	triton-inference-server/server	Production model serving with dynamic batching and multi-backend support.
MLC-LLM	mlc-ai/mlc-llm	Cross-platform LLM deployment through machine learning compilation.
DeepSpeed-MII [172]	deepspeedai/DeepSpeed-MII	High-performance LLM serving with DeepSpeed inference optimizations.
Sarathi-Serve [173]	microsoft/sarathi-serve	Research-oriented LLM serving with throughput-latency optimization.
WebLLM [174]	mlc-ai/web-llm	In-browser LLM inference with WebGPU acceleration.
FastChat [175]	lm-sys/FastChat	Chatbot training, evaluation, and multi-model serving platform.
Ray	ray-project/ray	Distributed runtime with scalable model serving through Ray Serve.

intelligence. LangChain [176] offers general interfaces for models, tools, retrievers, and integrations, which makes it a natural fit for building agentic vulnerability-analysis workflows that couple LLM reasoning with external evidence. LlamaIndex [177] focuses on linking external data to LLM applications through parsing, indexing, and retrieval (a design that suits constructing vulnerability knowledge bases from source code, security advisories, CVE descriptions, and project documentation). Haystack [178] supports modular retrieval, generation, and agent pipelines, enabling controllable RAG workflows for vulnerability triage and evidence-grounded question answering. DSPy [179] provides a programming model for modular language-model pipelines and optimization, proving valuable when RAG prompts, retrieval modules, or agent steps require systematic tuning for vulnerability-intelligence tasks. RAGFlow [180] delivers document-aware RAG with retrieval, reranking, citations, and agent capabilities, making it relevant to security reports and long vulnerability documents that demand grounded answers. Milvus [181] and Weaviate supply vector storage and semantic retrieval, supporting large-scale search over code embeddings, vulnerability descriptions, patch records, and security knowledge. Microsoft GraphRAG [182] and LightRAG [183] further extend RAG with graph-based retrieval, which helps preserve relations among software entities, vulnerabilities, CWEs, CVEs, commits, and patches. RAGAS [184] and RAGChecker [185] supply evaluation tooling for RAG systems, helping diagnose retrieval quality, generation faithfulness, and claim-level correctness. In practice, these frameworks cover the full spectrum: data parsing, indexing, vector storage, graph-based retrieval, agent orchestration, grounded generation, reranking, and evaluation. Due to space limitations, more frameworks and tools are not discussed in detail.

4.3.5. Integrated Pipeline

In a typical vulnerability detection system, a program analysis front-end such as Joern extracts program graphs; NetworkX cleans, slices, aligns, and converts them; and PyG or DGL performs message passing and vulnerability prediction. In parallel, Hugging Face Transformers encodes source-code tokens, and the resulting textual representations can be fused with graph representations. At inference time, vLLM, SGLang, or Ollama serves or locally executes LLMs, while LangChain or LlamaIndex orchestrates retrieval from Milvus or Weaviate to augment prompts with structured vulnerability knowledge. The result is a layered architecture that lets detection pipelines jointly exploit program structures, data and control dependencies, lexical context, historical vulnerability

Table 11. RAG computational frameworks for vulnerability intelligence.

Framework	GitHub Link	Main Role and Strength
LangChain [176]	langchain-ai/langchain	Builds agentic LLM applications with model, tool, retriever, and integration interfaces.
LlamaIndex [177]	run-llama/llama_index	Data framework for ingestion, indexing, retrieval, and generation.
Haystack [178]	deepset-ai/haystack	Builds modular retrieval, generation, and agent pipelines.
DSPy [179]	stanfordnlp/dspy	Programs and optimizes modular LM pipelines, RAG systems, and agents.
RAGFlow [180]	infiniflow/ragflow	Provides document-aware RAG with retrieval, reranking, citations, and agent support.
Milvus [181]	milvus-io/milvus	Provides scalable vector search and hybrid retrieval for embedding-based applications.
Weaviate	weaviate/weaviate	Combines vector storage, semantic search, filtering, and RAG support.
Microsoft GraphRAG [182]	microsoft/graphrag	Builds graph-based RAG pipelines from unstructured text.
LightRAG [183]	HKUDS/LightRAG	Provides lightweight graph-based RAG with vector and knowledge-graph retrieval.
RAGAS [184]	explodinggradients/ragas	Evaluates RAG and LLM applications with automated metrics and test data.
RAGChecker [185]	amazon-science/RAGChecker	Evaluates RAG systems with retriever, generator, and claim-level diagnostics.

records, and LLM-generated semantic reasoning. That provides an engineering foundation for scalable, evidence-grounded vulnerability analysis.

5. Challenges and Opportunities

- (a) **Poor Interpretability.** In graph-driven vulnerability analysis, model interpretability typically relies on correlation analysis, i.e., identifying which graph nodes, edges, or subgraphs exhibit the strongest statistical association with predictions. Representative methods include GNN-based explainers, feature attribution methods such as SHAP, and salient subgraph extraction. However, correlation-based explanations often fail to uncover the true causal mechanisms behind vulnerability formation. An exploitable vulnerability can stem from intricate interactions among long-range inter-procedural data flows, omitted security checks, and subtle environmental assumptions, whereas attribution methods may erroneously assign high importance to irrelevant identifiers or boilerplate code, thereby obscuring the real vulnerable points. Achieving causal interpretability demands that models comprehend how attacker-controllable data propagates through program dependence graphs, explain why missing specific control conditions leads to an exploitable state, and pinpoint which dependence chain a repair patch severs. Program-semantics-level explanations thus call for specialized methods, such as generating minimal vulnerability evidence subgraphs or cross-validating model predictions with static analysis, symbolic execution, and taint analysis. Such methods can elevate explanations from “where the model attends” to “why the vulnerability occurs.”
- (b) **Limited Cross-language Generalization.** Most existing graph-driven vulnerability detection models are trained on datasets from a single programming language. When applied to unseen languages, their detection performance often drops sharply. This degradation stems primarily from substantial differences across languages in syntactic structures, control-flow patterns, and standard library APIs, which cause models to capture language-specific surface statistics rather than language-agnostic vulnerability semantics. A promising direction is to learn language-independent program graph representations. Future work could design unified graph schemas that abstract away language-specific syntactic details, conduct multilingual joint pretraining to capture shared defect patterns across languages, and employ transfer learning to migrate vulnerability knowledge from resource-rich languages to new ones.

- (c) **Data Quality and Label Noise.** Existing vulnerability datasets commonly suffer from label noise, sample duplication, class imbalance, and missing context. Many real-world datasets are derived from CVE-fixing commits and simply label pre-patch code as vulnerable and post-patch code as non-vulnerable [186]. However, not all modified code in a commit is necessarily related to the vulnerability, while unmodified surrounding code may contain conditions necessary for triggering it. Such supervision can lead models to learn patch patterns, project-specific coding habits, or code templates instead of the semantic characteristics of vulnerabilities. A more reliable alternative is to replace coarse-grained binary labels with fine-grained vulnerability evidence annotations, such as vulnerable lines, critical variables, missing checks, and source-to-sink paths, coupled with patch context and verifiable test cases.
- (d) **Lack of Standardized and Fair Benchmarks.** Although many graph-driven vulnerability detection and localization methods have been proposed, standardized, fair, and comparable benchmarks remain scarce. Existing studies frequently rely on small-scale subsets or self-collected datasets and adopt inconsistent data splitting and filtering criteria. Evaluation is also mostly restricted to function-level binary classification accuracy, while fine-grained localization accuracy, explanation faithfulness, and repair effectiveness are largely neglected. This undermines the credibility and comparability of results across different models. Progress demands standardized evaluation protocols and large-scale benchmark datasets. Constructing multi-task, multilingual, multi-granularity vulnerability graph datasets would enable fair comparisons among graph representation learning methods, graph construction tools, and downstream reasoning strategies.
- (e) **New Security Challenges Introduced by LLM-generated Code and Agents.** As AI programming tools such as Codex, Claude Code, and OpenClaw are increasingly integrated into software development workflows, vulnerability risks are expanding from traditional “human-written code defects” to “AI-generated code and agent action-chain defects.” In AI-assisted development, code can be automatically generated, lightly modified by developers and directly merged, or even produced by agents that autonomously read repositories, invoke tools, modify configurations, install dependencies, execute tests, and submit patches [187, 188]. Consequently, vulnerabilities may stem not only from human coding mistakes but also from model generation bias, prompt contamination, context misunderstanding, unsafe tool invocation, dependency pollution, configuration hijacking, and incomplete automated repair patches. More importantly, agent-driven development elevates security concerns from the “code fragment level” to system-level trustworthiness across the “model–prompt–tool–configuration–environment–supply chain” stack. AI artifacts such as prompt templates, context files, tool invocation declarations, Model Context Protocol (MCP) configurations, and model configurations are gradually becoming new supply-chain attack surfaces. Meanwhile, when AI applications integrate web browsing, tool invocation, and external API access, traditional web vulnerabilities, such as Server-Side Request Forgery (SSRF), API key leakage, and unauthorized access, can be embedded into agent toolchains and further amplified if the backend lacks strict authentication, access control, input validation, and outbound request filtering. Therefore, software security research in the AI era should not focus only on detecting vulnerabilities in existing code but should further expand toward ensuring the end-to-end security of AI-generated, AI-modified, and AI-executed code. Graph-driven methods remain valuable in this direction: traditional AST, CFG, DFG, PDG, and CPG can capture code semantics, while action sequence graphs, tool invocation graphs, permission access graphs, configuration dependency graphs, and supply-chain propagation graphs can further model the complete behavior chain of agents. By combining program dependence graphs with agent behavior graphs, graph-driven vulnerability intelligence can evolve from single-point vulnerability detection into a security closed-loop governance framework for AI-native software engineering.

6. Conclusions

This survey reviews recent progress in program graph learning for software vulnerability detection, adopting the perspective of graph-driven vulnerability intelligence. We focus on key tasks (vulnerability detection, classification, localization, and repair) and systematically summarize representative graph learning methods, graph-based multimodal methods, LLM-assisted methods, vulnerability repair methods, as well as related datasets, program graph construction tools, and computing frameworks. Program graphs can explicitly capture syntactic structures, control flow, data flow, call relations, and program dependence information in code, providing a solid foundation for vulnerability semantic modeling and structured reasoning. This shift allows vulnerability analysis to move beyond shallow code-pattern recognition toward intelligent reasoning grounded in program semantics and security evidence. Although existing studies have made substantial progress, challenges persist: model interpretability, cross-language generalization, data quality, and benchmark standardization. At the same time, LLM-generated code and AI Agent-driven development are reshaping how software is built, and they bring new security concerns

(tool invocation errors, context contamination, unreliable automated patches, supply-chain risk propagation, among others). Future research, therefore, needs to bring together program graph modeling, neural representation learning, static and dynamic analysis, LLM reasoning, AI Agent collaboration, and external security knowledge more tightly, aiming to build reliable, transferable, explainable, and validation-aware vulnerability intelligence systems for intelligent software security scenarios.

Author Contributions

T.Y.: investigation, writing—original draft preparation; J.W.: data curation; M.L.: writing—review and editing; Y.H. and J.H.: software, validation; X.C.: methodology; W.H.: writing—review and editing, supervision; A.B.: writing—review and editing; All authors have read and agreed to the published version of the manuscript.

Funding

This research was funded by the National Natural Science Foundation of China (grant number 62476203), the Key Project of Traditional Chinese Medicine Joint Fund of Hubei Provincial Natural Science Foundation (grant number 2025AFD47), the Hubei Province Science and Technology Innovation Plan Project (grant number 2025BCB035), the Shenzhen Natural Science Foundation Project (grant numbers JCYJ20250604122534006 and JCYJ20230807090211021), and Guangdong Provincial Natural Science Foundation General Project (grant number 2025A1515012155).

Institutional Review Board Statement

Not applicable.

Informed Consent Statement

Not applicable.

Data Availability Statement

Not applicable.

Conflicts of Interest

The authors declare no conflict of interest.

Use of AI and AI-Assisted Technologies

No AI tools were utilized for this paper.

References

1. Shao, M.; Ding, Y. FVD-DPM: Fine-Grained Vulnerability Detection via Conditional Diffusion Probabilistic Models. In Proceedings of the USENIX Security Symposium, Philadelphia, PA, USA, 14–16 August 2024; pp. 7375–7392.
2. Jayalath, R.K.; Ahmad, H.; Goel, D.; et al. Microservice Vulnerability Analysis: A Literature Review with Empirical Insights. *IEEE Access* **2024**, *12*, 155168–155204. <https://doi.org/10.1109/access.2024.3481374>.
3. Liu, T.; Deng, Z.; Meng, G.; et al. Demystifying RCE Vulnerabilities in LLM-Integrated Apps. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Salt Lake City, UT, USA, 14–18 October 2024; pp. 1716–1730. <https://doi.org/10.1145/3658644.3690338>.
4. Choi, D.; Seo, H.; Kim, K.; et al. Uncovering Threats in Container Systems: A Study on Misconfigured Container Components in the Wild. *IEEE Access* **2024**, *12*, 192931–192945. <https://doi.org/10.1109/access.2024.3514751>.
5. Cost of a Data Breach Report 2024. Available online: <https://www.ibm.com/think/insights/whats-new-2024-cost-of-a-data-breach-report/> (accessed on 2 July 2026).
6. Sun, T.; He, N.; Xiao, J.; et al. All Your Tokens Are Belong to Us: Demystifying Address Verification Vulnerabilities in Solidity Smart Contracts. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 3567–3584.
7. Cybercrime To Cost The World \$10.5 Trillion Annually By 2025. Available online: <https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/> (accessed on 2 July 2026).
8. Gokkaya, B.; Aniello, L.; Halak, B. Software Supply Chain: Review of Attacks, Risk Assessment Strategies and Security Controls. *arXiv* **2023**, arXiv:2305.14157.
9. Williams, L.; Benedetti, G.; Hamer, S.; et al. Research Directions in Software Supply Chain Security. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–38. <https://doi.org/10.1145/3714464>.

10. Lin, J.; Zhang, H.; Adams, B.; et al. Vulnerability Management in Linux Distributions: An Empirical Study on Debian and Fedora. *Empir. Softw. Eng.* **2023**, *28*, 47. <https://doi.org/10.1007/s10664-022-10267-7>.
11. Shaon, M.S.H.; Akter, M.S. Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models. *Electronics* **2025**, *14*, 4449. <https://doi.org/10.3390/electronics14224449>.
12. Zhang, J.; Liu, Z.; Hu, X.; et al. Vulnerability Detection by Learning from Syntax-Based Execution Paths of Code. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4196–4212. <https://doi.org/10.1109/tse.2023.3286586>.
13. Deng, P.; Zhang, L.; Meng, Y.; et al. ChainFuzz: Exploiting Upstream Vulnerabilities in Open-Source Supply Chains. In Proceedings of the USENIX Security Symposium, Seattle, WA, USA, 13–15 August 2025; pp. 6199–6218.
14. Liao, Z.; Zheng, Z.; Chen, X.; et al. SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, 18–22 July 2022; pp. 752–764. <https://doi.org/10.1145/3533767.3534222>.
15. Liao, Z.; Nan, Y.; Liang, H.; et al. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proc. ACM Softw. Eng.* **2024**, *1*, 249–270. <https://doi.org/10.1145/3643738>.
16. Torres, A.; Costa, P.; Amaral, L.; et al. Runtime Verification of Crypto APIs: An Empirical Study. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4510–4525. <https://doi.org/10.1109/tse.2023.3301660>.
17. Johannesmeyer, B.; Iseman, R.; Giuffrida, C.; et al. Dynamic Detection of Vulnerable DMA Race Conditions. In Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, Taipei, Taiwan, 13–17 October 2025; pp. 4499–4513. <https://doi.org/10.1145/3719027.3765126>.
18. Ruaro, N.; Zeng, K.; Dresel, L.; et al. SyML: Guiding Symbolic Execution toward Vulnerable States through Pattern Learning. In Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, 6–8 October 2021; pp. 456–468. <https://doi.org/10.1145/3471621.3471865>.
19. So, S.; Hong, S.; Oh, H. SmartTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In Proceedings of the USENIX Security Symposium, Online, 11–13 August 2021; pp. 1361–1378.
20. Nagy, S.; Nguyen-Tuong, A.; Hiser, J.D.; et al. Same Coverage, Less Bloat: Accelerating Binary-Only Fuzzing with Coverage-Preserving Coverage-Guided Tracing. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Online, 15–19 November 2021; pp. 351–365. <https://doi.org/10.1145/3460120.3484787>.
21. Schloegel, M.; Bars, N.; Schiller, N.; et al. SoK: Prudent Evaluation Practices for Fuzzing. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2024; pp. 1974–1993. <https://doi.org/10.1109/sp54263.2024.00137>.
22. Fioraldi, A.; Maier, D.C.; Zhang, D.; et al. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 1051–1065. <https://doi.org/10.1145/3548606.3560602>.
23. Wu, B.; Zou, F. Code Vulnerability Detection Based on Deep Sequence and Graph Models: A Survey. *Secur. Commun. Netw.* **2022**, *2022*, 1176898. <https://doi.org/10.1155/2022/1176898>.
24. Harzevili, N.; Boaye Belle, A.; Wang, J.; et al. A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Comput. Surv.* **2024**, *57*, 1–36.
25. Qiu, F.; Liu, Z.; Hu, X.; et al. Vulnerability Detection via Multiple-Graph-Based Code Representation. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2178–2199. <https://doi.org/10.1109/tse.2024.3427815>.
26. Wen, X.C.; Chen, Y.; Gao, C.; et al. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, VIC, Australia, 14–20 May 2023; pp. 2275–2286. <https://doi.org/10.1109/icse48619.2023.00191>.
27. Ghaffarian, S.M.; Shahriari, H.R. Neural Software Vulnerability Analysis Using Rich Intermediate Graph Representations of Programs. *Inf. Sci.* **2021**, *553*, 189–207.
28. Katsadourous, E.; Patrikakis, C. A Survey on Vulnerability Prediction Using GNNs. In Proceedings of the 26th Pan-Hellenic Conference on Informatics, Athens, Greece, 25–27 November 2022; pp. 38–43. <https://doi.org/10.1145/3575879.3575964>.
29. bin Mohd'Illzam, M.I.E.; Yong, K.S.C.; Then, P.H.H.; et al. Comparative Analysis of Graph Representation Techniques for Code Vulnerability Detection. In Proceedings of the 2023 International Conference on Engineering and Emerging Technologies (ICEET), Istanbul, Turkiye, 27–28 October 2023; pp. 1–6.
30. Lin, C.; Xu, Y.; Fang, Y.; et al. VulEye: A Novel Graph Neural Network Vulnerability Detection Approach for PHP Application. *Appl. Sci.* **2023**, *13*, 825. <https://doi.org/10.3390/app13020825>.
31. Xiao, W.; Hou, Z.; Wang, T.; et al. MSGVUL: Multi-Semantic Integration Vulnerability Detection Based on Relational Graph Convolutional Neural Networks. *Inf. Softw. Technol.* **2024**, *170*, 107442. <https://doi.org/10.1016/j.infsof.2024.107442>.
32. Cheng, B.; Wang, K.; Gao, C.; et al. SliceLocator: Locating Vulnerable Statements with Graph-Based Detectors. *arXiv* **2024**, arXiv:2401.02737.
33. Mirsky, Y.; Macon, G.; Brown, M.; et al. VulChecker: Graph-Based Vulnerability Localization in Source Code. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 6557–6574.

34. Cheng, B.; Zhao, S.; Wang, K.; et al. Beyond Fidelity: Explaining Vulnerability Localization of Learning-Based Detectors. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 127:1–127:33. <https://doi.org/10.1145/3641543>.
35. Li, L.; Ding, S.H.H.; Tian, Y.; et al. VulANalyzeR: Explainable Binary Vulnerability Detection with Multi-Task Learning and Attentional Graph Convolution. *ACM Trans. Priv. Secur.* **2023**, *26*, 1–25. <https://doi.org/10.1145/3585386>.
36. Mao, Q.; Li, Z.; Hu, X.; et al. Towards Explainable Vulnerability Detection with Large Language Models. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2957–2971. <https://doi.org/10.1109/tse.2025.3605442>.
37. Fu, M.; Tantithamthavorn, C. Linevul: A Transformer-Based Line-Level Vulnerability Prediction. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 608–620. <https://doi.org/10.1145/3524842.3528452>.
38. Zhen, Z.; Zhao, X.; Zhang, J.; et al. DA-GNN: A Smart Contract Vulnerability Detection Method Based on Dual Attention Graph Neural Network. *Comput. Netw.* **2024**, *242*, 110238. <https://doi.org/10.1016/j.comnet.2024.110238>.
39. Li, Y.; Shezan, F.H.; Wei, B.; et al. SoK: Towards Effective Automated Vulnerability Repair. In Proceedings of the USENIX Security 25, Seattle, WA, USA, 13–15 August 2025; pp. 4441–4462.
40. Zhang, Q.; Fang, C.; Xie, Y.; et al. A Systematic Literature Review on Large Language Models for Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* **2026**. <https://doi.org/10.1145/3799693>.
41. Zhu, C.; Wei, R.; Chen, L.; et al. Vulnerability Localization Based on Intermediate Code Representation and Feature Fusion. *Comput. J.* **2024**, *67*, 2749–2762. <https://doi.org/10.1093/comjnl/bxae041>.
42. Zhou, X.; Cao, S.; Sun, X.; et al. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–31. <https://doi.org/10.1145/3708522>.
43. Zhou, Y.; Liu, S.; Siow, J.; et al. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.
44. Wang, H.; Ye, G.; Tang, Z.; et al. Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 1943–1958. <https://doi.org/10.1109/tifs.2020.3044773>.
45. Chakraborty, S.; Krishna, R.; Ding, Y.; et al. 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>.
46. Nguyen, V.A.; Nguyen, D.Q.; Nguyen, V.; et al. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings, Pittsburgh, PA, USA, 22–24 May 2022; pp. 178–182. <https://doi.org/10.1109/icse-companion55297.2022.9793807>.
47. Qian, P.; Liu, Z.; Yin, Y.; et al. Cross-Modality Mutual Learning for Enhancing Smart Contract Vulnerability Detection on Bytecode. In Proceedings of the ACM Web Conference 2023, Austin, TX, USA, 30 April–4 May 2023; pp. 2220–2229. <https://doi.org/10.1145/3543507.3583367>.
48. Mikolov, T.; Sutskever, I.; Chen, K.; et al. Distributed Representations of Words and Phrases and Their Compositionality. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 3111–3119.
49. Li, Y.; Wang, S.; Nguyen, T.N. Vulnerability Detection with Fine-Grained Interpretations. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 23–28 August 2021; pp. 292–303. <https://doi.org/10.1145/3468264.3468597>.
50. Pennington, J.; Socher, R.; Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543. <https://doi.org/10.3115/v1/d14-1162>.
51. Cheng, X.; Wang, H.; Hua, J.; et al. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 1–33. <https://doi.org/10.1145/3436877>.
52. Le, Q.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the International Conference on Machine Learning, Beijing, China, 22–24 June 2014; pp. 1188–1196.
53. Cao, S.; Sun, X.; Bo, L.; et al. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* **2021**, *136*, 106576. <https://doi.org/10.1016/j.infsof.2021.106576>.
54. Hin, D.; Kan, A.; Chen, H.; et al. LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks. In Proceedings of the MSR '22: 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 596–607. <https://doi.org/10.1145/3524842.3527949>.
55. Feng, Z.; Guo, D.; Tang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Association for Computational Linguistics: EMNLP 2020, Online, 16–20 November 2020; pp. 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>.
56. Guo, D.; Ren, S.; Lu, S.; et al. GraphCodeBERT: Pre-Training Code Representations with Data Flow. *arXiv* **2020**, arXiv:2009.08366.
57. Nguyen, H.V.; Zheng, J.; Inomata, A.; et al. Code Aggregate Graph: Effective Representation for Graph Neural Networks to Detect Vulnerable Code. *IEEE Access* **2022**, *10*, 123786–123800. <https://doi.org/10.1109/access.2022.3216395>.
58. Guo, W.; Fang, Y.; Huang, C.; et al. HyVulDect: A Hybrid Semantic Vulnerability Mining System Based on Graph Neural Network. *Comput. Secur.* **2022**, *121*, 102823. <https://doi.org/10.1016/j.cose.2022.102823>.
59. Wen, X.C.; Gao, C.; Ye, J.; et al. 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>.
60. Ge, K.; Han, Q.B. Hidden Code Vulnerability Detection: A Study of the Graph-BiLSTM Algorithm. *Inf. Softw. Technol.* **2024**, *175*, 107544. <https://doi.org/10.1016/j.infsof.2024.107544>.
 61. Pagliardini, M.; Gupta, P.; Jaggi, M. Unsupervised Learning of Sentence Embeddings Using Compositional N-Gram Features. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, New Orleans, LA, USA, 1–6 June 2018; pp. 528–540. <https://doi.org/10.18653/v1/n18-1049>.
 62. Xiao, P.; Xiao, Q.; Zhang, X.; et al. 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>.
 63. Hao, J.; Kwon, Y.W. Enhancing Graph-Based Vulnerability Detection through Standardized Deep Learning Pipelines. In Proceedings of the 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Sanya, China, 17–21 December 2024; pp. 1231–1238. <https://doi.org/10.1109/trustcom63139.2024.00174>.
 64. Huang, Y.; He, M.; Wang, X.; et al. 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>.
 65. Ling, M.; Tang, M.; Bian, D.; et al. A Dual Graph Neural Networks Model Using Sequence Embedding as Graph Nodes for Vulnerability Detection. *Inf. Softw. Technol.* **2025**, *177*, 107581. <https://doi.org/10.1016/j.infsof.2024.107581>.
 66. Shao, M.; Ding, Y.; Cao, J.; et al. GraphFVD: Property Graph-Based Fine-Grained Vulnerability Detection. *Comput. Secur.* **2025**, *151*, 104350.
 67. Sun, X.; Zhou, M.; Cao, S.; et al. HgtJIT: Just-in-Time Vulnerability Detection Based on Heterogeneous Graph Transformer. *IEEE Trans. Dependable Secur. Comput.* **2025**, *22*, 6522–6538. <https://doi.org/10.1109/tdsc.2025.3586669>.
 68. Lin, B.; Wang, S.; Liu, Z.; et al. CCT5: A Code-Change-Oriented Pre-Trained Model. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 3–9 December 2023; pp. 1509–1521.
 69. Nguyen, S.; Nguyen, T.T.; Vu, T.T.; et al. Code-Centric Learning-Based Just-in-Time Vulnerability Detection. *J. Syst. Softw.* **2024**, *214*, 112014. <https://doi.org/10.1016/j.jss.2024.112014>.
 70. Piao, Y.; Zhang, Z.; Kong, Z. Code Vulnerability Detection Method Based on PreTrained Language Model and Gating Graph Neural Network. In Proceedings of the 2025 4th International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE), Chengdu, China, 24–26 October 2025; pp. 685–688. <https://doi.org/10.1109/cbase67452.2025.11335562>.
 71. Zhuang, Y.; Liu, Z.; Qian, P.; et al. Smart Contract Vulnerability Detection Using Graph Neural Network. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, Online, 7–15 January 2021; pp. 3283–3290. <https://doi.org/10.24963/ijcai.2020/454>.
 72. Ni, C.; Yin, X.; Li, X.; et al. Abundant Modalities Offer More Nutrients: Multi-Modal-Based Function-Level Vulnerability Detection. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 33:1–33:31. <https://doi.org/10.1145/3731557>.
 73. Cao, X.; Wang, J.; Wu, P.; et al. VulMPFF: A Vulnerability Detection Method for Fusing Code Features in Multiple Perspectives. *IET Inf. Secur.* **2024**, *2024*, 4313185. <https://doi.org/10.1049/2024/4313185>.
 74. Lu, G.; Ju, X.; Chen, X.; et al. GRACE: Empowering LLM-Based Software Vulnerability Detection with Graph Structure and in-Context Learning. *J. Syst. Softw.* **2024**, *212*, 112031.
 75. Chu, H.; Zhang, P.; Dong, H.; et al. Deepfusion: Smart Contract Vulnerability Detection via Deep Learning and Data Fusion. *IEEE Trans. Reliab.* **2025**, *74*, 3544–3558.
 76. Lekssays, A.; Mouhcine, H.; Tran, K.; et al. LLMxCPG: Context-Aware Vulnerability Detection through Code Property Graph-Guided Large Language Models. In Proceedings of the USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 489–507.
 77. Cao, Y.; Ju, X.; Chen, X.; et al. MCL-VD: Multi-Modal Contrastive Learning with LoRA-Enhanced GraphCodeBERT for Effective Vulnerability Detection. *Autom. Softw. Eng.* **2025**, *32*, 67. <https://doi.org/10.1007/s10515-025-00543-3>.
 78. Liu, R.; Wang, Y.; Xu, H.; et al. 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>.
 79. Cai, W.; Chen, J.; Yu, J.; et al. A Software Vulnerability Detection Method Based on Multi-Modality with Unified Processing. *Inf. Softw. Technol.* **2025**, *182*, 107703. <https://doi.org/10.1016/j.infsof.2025.107703>.
 80. Li, Q.; Jiang, G.; He, P.; et al. CMF-Vul: Advancing Automated Vulnerability Detection via Contrastive Multimodal Fusion and Challenge-Driven Representation Learning. *Empir. Softw. Eng.* **2026**, *31*, 152. <https://doi.org/10.1007/s10664-026-10886-4>.
 81. Zhang, G.; Yao, T.; Qin, J.; et al. CodeSAGE: A Multi-Feature Fusion Vulnerability Detection Approach Using Code Attribute Graphs and Attention Mechanisms. *J. Inf. Secur. Appl.* **2025**, *89*, 103973. <https://doi.org/10.1016/j.jisa.2025.103973>.
 82. Hou, A.; Su, B.; Niu, W.; et al. CPGHunter: LLM-Guided Semantic Modeling for Scalable Vulnerability Detection via Taint Analysis. *Empir. Softw. Eng.* **2026**, *31*, 106. <https://doi.org/10.1007/s10664-026-10842-2>.
 83. Zhu, H.; Li, J.; Gao, C.; et al. Specification-Guided Vulnerability Detection with Large Language Models. *arXiv* **2025**, arXiv:2511.04014.

84. Du, X.; Zheng, G.; Wang, K.; et al. Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG. *ACM Trans. Softw. Eng. Methodol.* **2026**. <https://doi.org/10.1145/3797277>.
85. Fayyazi, R.; Trueba, S.H.; Zuzak, M.; et al. ProveRAG: Provenance-Driven Vulnerability Analysis with Automated Retrieval-Augmented LLMs. *IEEE Access* **2025**, *13*, 212815–212826. <https://doi.org/10.1109/access.2025.3638251>.
86. Daneshvar, S.S.; Nong, Y.; Yang, X.; et al. VulScribeR: Exploring RAG-Based Vulnerability Augmentation with LLMs. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–26.
87. Quynh Nhu, N.D.; Minh Quan, L.; Van, T.H.; et al. RAG-SmartVuln: Enhancing Smart Contract Vulnerability Detection via Retrieval-Augmented LLMs. In Proceedings of the 2025 International Conference on Multimedia Analysis and Pattern Recognition (MAPR), Nha Trang, Vietnam, 14–15 August 2025; pp. 1–6. <https://doi.org/10.1109/mapr67746.2025.11134018>.
88. Chen, Y. AutoReview: An LLM-Based Multi-Agent System for Security Issue-Oriented Code Review. In Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, Trondheim, Norway, 23–27 June 2025; pp. 1022–1024. <https://doi.org/10.1145/3696630.3728618>.
89. Widyasari, R.; Weyssow, M.; Irsan, I.C.; et al. Let the Trial Begin: A Mock-Court Approach to Vulnerability Detection Using LLM-Based Agents. *arXiv* **2025**, arXiv:2505.10961.
90. Jie, W.; Qiu, W.; Yang, H.; et al. Agent4Vul: Multimodal LLM Agents for Smart Contract Vulnerability Detection. *Sci. China Inf. Sci.* **2025**, *68*, 160101. <https://doi.org/10.1007/s11432-024-4402-2>.
91. Ghaleb, A.; Pattabiraman, K. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, 18–22 July 2020; pp. 415–427. <https://doi.org/10.1145/3395363.3397385>.
92. Hu, S.; Huang, T.; İlhan, F.; et al. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. In Proceedings of the 2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Atlanta, GA, USA, 1–4 November 2023; pp. 297–306. <https://doi.org/10.1109/tps-isa58951.2023.00044>.
93. Liu, H.; Einstein, L.; Yang, J.; et al. SecureForge: Finding and Preventing Vulnerabilities in LLM-Generated Code via Prompt Optimization. *arXiv* **2026**, arXiv:2605.08382.
94. Baumann, J.; Padmakumar, V.; Li, X.; et al. SWE-Chat: Coding Agent Interactions from Real Users in the Wild. *arXiv* **2026**, arXiv:2604.20779.
95. Li, X.; Su, Y.; Liu, J.; et al. VULSOLVER: Vulnerability Detection via LLM-Driven Constraint Solving. *arXiv* **2025**, arXiv:2509.00882.
96. OWASP Foundation. OWASP Benchmark Project. Available online: <https://owasp.org/www-project-benchmark/> (accessed on 2 July 2026).
97. Sun, Y.; Wu, D.; Xue, Y.; et al. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13. <https://doi.org/10.1145/3597503.3639117>.
98. Yi, X.; Fang, Y.; Wu, D.; et al. BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects. *arXiv* **2022**, arXiv:2208.00205.
99. Zhang, Z.; Zhang, B.; Xu, W.; et al. Demystifying Exploitable Bugs in Smart Contracts. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering, Melbourne, VIC, Australia, 14–20 May 2023; pp. 615–627. <https://doi.org/10.1109/icse48619.2023.00061>.
100. Jiang, J.; Xiong, Y.; Zhang, H.; et al. Shaping Program Repair Space with Existing Patches and Similar Code. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 298–309. <https://doi.org/10.1145/3213846.3213871>.
101. Liu, K.; Koyuncu, A.; Kim, D.; et al. TBar: Revisiting Template-Based Automated Program Repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 31–42. <https://doi.org/10.1145/3293882.3330577>.
102. Koyuncu, A.; Liu, K.; Bissyandé, T.F.; et al. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empir. Softw. Eng.* **2020**, *25*, 1980–2024. <https://doi.org/10.1007/s10664-019-09780-z>.
103. Wei, Y.; Bo, L.; Wu, X.; et al. VulRep: Vulnerability Repair Based on Inducing Commits and Fixing Commits. *EURASIP J. Wirel. Commun. Netw.* **2023**, *2023*, 34. <https://doi.org/10.1186/s13638-023-02242-7>.
104. Dinella, E.; Dai, H.; Li, Z.; et al. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In Proceedings of the International Conference on Learning Representations, Online, 26–30 April 2020.
105. Zhu, Q.; Sun, Z.; Xiao, Y.A.; et al. A Syntax-Guided Edit Decoder for Neural Program Repair. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 23–28 August 2021; pp. 341–353.
106. Wang, R.; Li, Z.; Gao, C.; et al. SPVR: Syntax-to-Prompt Vulnerability Repair Based on Large Language Models. *Autom. Softw. Eng.* **2026**, *33*, 38. <https://doi.org/10.1007/s10515-025-00579-5>.
107. Chi, J.; Qu, Y.; Liu, T.; et al. Seqtrans: Automatic Vulnerability Fix via Sequence to Sequence Learning. *IEEE Trans. Softw. Eng.* **2023**, *49*, 564–585.

108. Chen, Z.; Komrmusch, S.; Tufano, M.; et al. Sequencer: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1943–1959.
109. Lutellier, T.; Pham, H.V.; Pang, L.; et al. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, 18–22 July 2020; pp. 101–114. <https://doi.org/10.1145/3395363.3397369>.
110. Chen, Z.; Komrmusch, S.; Monperrus, M. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Trans. Softw. Eng.* **2023**, *49*, 147–165.
111. Fu, M.; Tantithamthavorn, C.; Le, T.; et al. Vulrepair: A T5-Based Automated Software Vulnerability Repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 935–947. <https://doi.org/10.1145/3540250.3549098>.
112. Zhou, X.; Kim, K.; Xu, B.; et al. Out of Sight, out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1071–1083. <https://doi.org/10.1145/3597503.3639222>.
113. Wang, C.; Zhang, J.; Gao, J.; et al. ContractTinker: Llm-Empowered Vulnerability Repair for Real-World Smart Contracts. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Sacramento, CA, USA, 27 October–1 November 2024; pp. 2350–2353. <https://doi.org/10.1145/3691620.3695349>.
114. Xia, C.S.; Wei, Y.; Zhang, L. Practical Program Repair in the Era of Large Pre-Trained Language Models. *arXiv* **2022**, arXiv:2210.14179.
115. Xia, C.S.; Zhang, L. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each Using ChatGPT. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Vienna, Austria, 16–20 September 2024; pp. 819–831. <https://doi.org/10.1145/3650212.3680323>.
116. Wang, W.; Wang, Y.; Joty, S.; et al. Rap-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 3–9 December 2023; pp. 146–158. <https://doi.org/10.1145/3611643.3616256>.
117. Jin, M.; Shahriar, S.; Tufano, M.; et al. InferFix: End-to-End Program Repair with LLMs. In Proceedings of the ESEC/FSE’23: 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 3–9 December 2023; pp. 1646–1656. <https://doi.org/10.1145/3611643.3613892>.
118. Liu, Z.; Ma, Y.; Xu, J.; et al. Agent that Debugs: Dynamic State-Guided Vulnerability Repair. *arXiv* **2025**, arXiv:2504.07634.
119. Juliet C/C++ 1.3 Test Suite (SARD Test Suite #112). Available online: <https://samate.nist.gov/SARD/test-suites/112> (accessed on 26 May 2026).
120. Fan, J.; Li, Y.; Wang, S.; et al. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Korea, 29–30 June 2020; pp. 508–512. <https://doi.org/10.1145/3379597.3387501>.
121. Bhandari, G.; Naseer, A.; Moonen, L. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, Online, 19–20 August 2021; pp. 30–39. <https://doi.org/10.1145/3475960.3475985>.
122. Wang, X.; Hu, R.; Gao, C.; et al. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, Lisbon, Portugal, 14–20 April 2024; pp. 472–483. <https://doi.org/10.1145/3639478.3647634>.
123. Nikitopoulos, G.; Dritsa, K.; Louridas, P.; et al. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 23–27 August 2021; pp. 1565–1569. <https://doi.org/10.1145/3468264.3473122>.
124. Li, Y.; Zhang, T.; Widyasari, R.; et al. CleanVul: Automatic Function-Level Vulnerability Detection in Code Commits Using LLM Heuristics. *arXiv* **2024**, arXiv:2411.17274.
125. Russell, R.; Kim, L.; Hamilton, L.; et al. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; pp. 757–762. <https://doi.org/10.1109/icmla.2018.00120>.
126. Li, Z.; Zou, D.; Xu, S.; et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *arXiv* **2018**, arXiv:1801.01681.
127. Ferreira, J.F.; Cruz, P.; Durieux, T.; et al. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Online, 21–25 September 2020; pp. 1349–1352. <https://doi.org/10.1145/3324884.3415298>.
128. Li, Z.; Zou, D.; Xu, S.; et al. 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>.
129. Li, Z.; Zou, D.; Xu, S.; et al. 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>.

130. Chen, Y.; Ding, Z.; Alowain, L.; et al. Diversevul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, Hong Kong, China, 16–18 October 2023; pp. 654–668. <https://doi.org/10.1145/3607199.3607242>.
131. Ni, C.; Shen, L.; Yang, X.; et al. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In Proceedings of the 21st International Conference on Mining Software Repositories, Lisbon, Portugal, 15–16 April 2024; pp. 738–742. <https://doi.org/10.1145/3643991.3644886>.
132. Ding, Y.; Fu, Y.; Ibrahim, O.; et al. Vulnerability Detection with Code Language Models: How Far Are We? *arXiv* **2024**, arXiv:2403.18624.
133. Lu, C.; Li, T.; Dehaene, T.; et al. ICVul: A Well-Labeled C/C++ Vulnerability Dataset with Comprehensive Metadata and VCCS. In Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada, 28–29 April 2025; pp. 154–158. <https://doi.org/10.1109/msr66628.2025.00034>.
134. Li, Z.; Dutta, S.; Naik, M. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In Proceedings of the International Conference on Learning Representations, Singapore, 24–28 April 2025; pp. 35735–35758.
135. Joern—The Bug Hunter’s Workbench. Available online: <https://joern.io/> (accessed on 1 June 2026).
136. CodeQL. Available online: <https://codeql.github.com/> (accessed on 1 June 2026).
137. Tree-Sitter. Available online: <https://tree-sitter.github.io/tree-sitter/> (accessed on 1 June 2026).
138. ANTLR: ANother Tool for Language Recognition. Available online: <https://www.antlr.org/> (accessed on 1 June 2026).
139. The LLVM Compiler Infrastructure. Available online: <https://llvm.org/> (accessed on 1 June 2026).
140. Frama-C: Framework for Modular Analysis of C Programs. Available online: <https://frama-c.com/> (accessed on 1 June 2026).
141. Available online: <https://wala.github.io/> (accessed on 1 June 2026).
142. Soot—A Framework for Analyzing and Transforming Java and Android Applications. Available online: <https://soot-oss.github.io/soot/> (accessed on 1 June 2026).
143. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 27 May 2019; pp. 8–15. <https://doi.org/10.1109/wetseb.2019.00008>.
144. Paszke, A.; Gross, S.; Massa, F.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019.
145. Fey, M.; Lenssen, J.E. Fast Graph Representation Learning with PyTorch Geometric. *arXiv* **2019**, arXiv:1903.02428.
146. Wang, M.; Zheng, D.; Ye, Z.; et al. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv* **2019**, arXiv:1909.01315.
147. Hagberg, A.; Swart, P.J.; Schult, D.A. et al. Exploring Network Structure, Dynamics, and Function Using NetworkX. In Proceedings of the 7th Python in Science Conference (SciPy 2008), Pasadena, CA, USA, August 19–24, 2008 ; pp. 11–15.
148. Hu, J.; Qian, S.; Fang, Q.; et al. Efficient Graph Deep Learning in Tensorflow with tf-geometric. In Proceedings of the 29th ACM International Conference on Multimedia, Chengdu, China, 20–24 October 2021; pp. 3775–3778. <https://doi.org/10.1145/3474085.3478322>.
149. Ma, Y.; Yu, D.; Wu, T.; et al. PaddlePaddle: An Open-Source Deep Learning Platform from Industrial Practice. *Front. Data Comput.* **2019**, *1*, 105–115. <https://doi.org/10.11871/jfdc.issn.2096.742x.2019.01.011>.
150. Cen, Y.; Hou, Z.; Wang, Y.; et al. CogDL: A Comprehensive Library for Graph Deep Learning. In Proceedings of the ACM Web Conference 2023, Austin, TX, USA, 30 April–4 May 2023; pp. 747–758. <https://doi.org/10.1145/3543507.3583472>.
151. Liu, M.; Luo, Y.; Wang, L.; et al. DIG: A Turnkey Library for Diving into Graph Deep Learning Research. *J. Mach. Learn. Res.* **2021**, *22*, 1–9.
152. Yang, H. AliGraph: A Comprehensive Graph Neural Network Platform. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage, AK, USA, 4–8 August 2019; pp. 3165–3166. <https://doi.org/10.1145/3292500.3340404>.
153. Euler: A Distributed Graph Deep Learning Framework. Available online: <https://github.com/alibaba/euler> (accessed on 6 June 2026).
154. Jia, Z.; Lin, S.; Gao, M.; et al. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. *Proc. Mach. Learn. Syst.* **2020**, *2*, 187–198.
155. Fan, W.; He, T.; Lai, L.; et al. GraphScope. *Proc. VLDB Endow.* **2021**, *14*, 2879–2892. <https://doi.org/10.14778/3476311.3476369>.
156. Pan, Z.; Wu, T.; Zhao, Q.; et al. GeaFlow: A Graph Extended and Accelerated Dataflow System. *Proc. ACM Manag. Data* **2023**, *1*, 1–27. <https://doi.org/10.1145/3589771>.
157. Fender, A.; Rees, B.; Eaton, J. RAPIDS cuGraph. In *Massive Graph Analytics*; Chapman and Hall/CRC: Boca Raton, FL, USA, 2022; pp. 483–493. <https://doi.org/10.1201/9781003033707-22>.
158. Zhu, Z.; Xu, S.; Tang, J.; et al. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In Proceedings of the WWW’19: The Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 2494–2504. <https://doi.org/10.1145/3308558.3313508>.
159. Wolf, T.; Debut, L.; Sanh, V.; et al. Transformers: State-of-the-Art Natural Language Processing. In Proceedings of

- the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Online, 16–20 November 2020; pp. 38–45. <https://doi.org/10.18653/v1/2020.emnlp-demos.6>.
160. Pfeiffer, J.; Rücklé, A.; Poth, C.; et al. Adapterhub: A Framework for Adapting Transformers. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Online, 16–20 November 2020; pp. 46–54. <https://doi.org/10.18653/v1/2020.emnlp-demos.7>.
 161. Gugger, S.; Debut, L.; Wolf, T.; et al. Accelerate: Training and Inference at Scale Made Simple, Efficient and Adaptable. Available online: <https://github.com/huggingface/accelerate> (accessed on 2 July 2026).
 162. Ott, M.; Edunov, S.; Baevski, A.; et al. Fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In Proceedings of the NAACL-HLT 2019: Demonstrations, Minneapolis, MN, USA, 2–7 June 2019. <https://doi.org/10.18653/v1/n19-4009>.
 163. Klein, G.; Kim, Y.; Deng, Y.; et al. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In Proceedings of the ACL 2017, System Demonstrations, Vancouver, BC, Canada, 30 July–4 August 2017; pp. 67–72. <https://doi.org/10.18653/v1/p17-4012>.
 164. Vaswani, A.; Bengio, S.; Brevdo, E.; et al. Tensor2Tensor for Neural Machine Translation. In Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track), Boston, MA, USA, 17–21 March 2018; pp. 193–199.
 165. Junczys-Dowmunt, M.; Grundkiewicz, R.; Dwojak, T.; et al. Marian: Fast Neural Machine Translation in C++. In Proceedings of the ACL 2018, System Demonstrations, Melbourne, Australia, 15–20 July 2018; pp. 116–121. <https://doi.org/10.18653/v1/p18-4020>.
 166. Hieber, F.; Denkowski, M.; Domhan, T.; et al. Sockeye 3: Fast Neural Machine Translation with PyTorch. *arXiv* **2022**, arXiv:2207.05851.
 167. Shoyebi, M.; Patwary, M.; Puri, R.; et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv* **2019**, arXiv:1909.08053.
 168. Rasley, J.; Rajbhandari, S.; Ruwase, O.; et al. Deepspeed: System Optimizations Enable Training Deep Learning Models with over 100 Billion Parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Virtual Event, CA, USA, 23–27 August 2020; pp. 3505–3506. <https://doi.org/10.1145/3394486.3406703>.
 169. Li, S.; Liu, H.; Bian, Z.; et al. Colossal-AI: A Unified Deep Learning System for Large-Scale Parallel Training. In Proceedings of the 52nd International Conference on Parallel Processing, Salt Lake City, UT, USA, 7–10 August 2023; pp. 766–775. <https://doi.org/10.1145/3605573.3605613>.
 170. Kwon, W.; Li, Z.; Zhuang, S.; et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. In Proceedings of the 29th ACM Symposium on Operating Systems Principles, Koblenz, Germany, 23–26 October 2023; pp. 611–626. <https://doi.org/10.1145/3600006.3613165>.
 171. Zheng, L.; Yin, L.; Xie, Z.; et al. SGLang: Efficient Execution of Structured Language Model Programs. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 62557–62583. <https://doi.org/10.52202/079017-2000>.
 172. Aminabadi, R.Y.; Rajbhandari, S.; Awan, A.A.; et al. Deepspeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In Proceedings of the SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 13–18 November 2022; pp. 1–15. <https://doi.org/10.1109/sc41404.2022.00051>.
 173. Agrawal, A.; Kedia, N.; Panwar, A.; et al. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, USA, 10–12 July 2024; pp. 117–134.
 174. Ruan, C.F.; Qin, Y.; Parthasarathy, A.R.; et al. WebLLM: A High-Performance in-Browser LLM Inference Engine. *arXiv* **2024**, arXiv:2412.15803.
 175. Zheng, L.; Chiang, W.L.; Sheng, Y.; et al. Judging Llm-as-a-Judge with Mt-Bench and Chatbot Arena. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 46595–46623. <https://doi.org/10.52202/075280-2020>.
 176. The Open Agent Engineering Ecosystem. Available online: <https://docs.langchain.com/> (accessed on 4 June 2026).
 177. Welcome to LlamaIndex. Available online: <https://docs.llamaindex.ai/> (accessed on 4 June 2026).
 178. Open-Source AI Orchestration for Production-Grade Agents. Available online: <https://haystack.deepset.ai/> (accessed on 4 June 2026).
 179. Khattab, O.; Singhvi, A.; Maheshwari, P.; et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv* **2023**, arXiv:2310.03714.
 180. Build a Superior Context Layer for AI Agents. Available online: <https://ragflow.io/> (accessed on 4 June 2026).
 181. Wang, J.; Yi, X.; Guo, R.; et al. Milvus: A Purpose-Built Vector Data Management System. In Proceedings of the 2021 International Conference on Management of Data, Online, 20–25 June 2021; pp. 2614–2627. <https://doi.org/10.1145/3448016.3457550>.
 182. Edge, D.; Trinh, H.; Cheng, N.; et al. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. *arXiv* **2024**, arXiv:2404.16130.
 183. Guo, Z.; Xia, L.; Yu, Y.; et al. LightRAG: Simple and Fast Retrieval-Augmented Generation. *arXiv* **2024**, arXiv:2410.05779.

184. Es, S.; James, J.; Espinosa Anke, L.; et al. Ragas: Automated Evaluation of Retrieval Augmented Generation. In Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations, St. Julians, Malta, 21–22 March 2024; pp. 150–158. <https://doi.org/10.18653/v1/2024.eacl-demo.16>.
185. Ru, D.; Qiu, L.; Hu, X.; et al. RAGChecker: A Fine-Grained Framework for Diagnosing Retrieval-Augmented Generation. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 21999–22027. <https://doi.org/10.52202/079017-0692>.
186. Guo, Y.; Bettaieb, S.; Casino, F. A Comprehensive Analysis on Software Vulnerability Detection Datasets: Trends, Challenges, and Road Ahead. *Int. J. Inf. Secur.* **2024**, *23*, 3311–3327. <https://doi.org/10.1007/s10207-024-00888-y>.
187. Perry, N.; Srivastava, M.; Kumar, D.; et al. Do Users Write More Insecure Code with Ai Assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, 26–30 November 2023; pp. 2785–2799. <https://doi.org/10.1145/3576915.3623157>.
188. Pearce, H.; Ahmad, B.; Tan, B.; et al. Asleep at the Keyboard? Assessing the Security of Github Copilot’s Code Contributions. *Commun. ACM* **2025**, *68*, 96–105. <https://doi.org/10.1145/3610721>.