Innovative Journal of Computing, Security and Digital Technologies
Volume 1, Issue 2, May – Aug 2026
Page no : 21-29

An Intelligent Framework for Automated Software Vulnerability Detection and Secure Code Generation Using Large Language Models

Author : Ravi Varman

Abstract

Large language models (LLMs) are increasingly used to explain source code, identify defects, generate programs, and propose software repairs. Their ability to interpret natural-language requirements and broad code context creates opportunities for security-aware development; however, syntactically correct and functionally plausible code is not necessarily secure. LLMs may overlook data-flow dependencies, generate insecure constructs, misclassify vulnerabilities, or recommend patches that introduce regressions. This review examines research on LLM-assisted vulnerability detection, secure code generation, and automated vulnerability repair. It synthesizes findings from recent surveys, empirical evaluations, benchmark studies, and foundational vulnerability datasets. The literature indicates that LLMs are promising as contextual reasoning and triage components, but their reliability varies with programming language, vulnerability class, input representation, prompt design, and evaluation method. Purely generative approaches are therefore insufficient for high-assurance use. The paper proposes a conceptual hybrid framework combining code-context preparation, LLM reasoning, deterministic static and dynamic analysis, patch validation, and human approval. It also identifies research priorities in benchmark reproducibility, multi-language coverage, explainability, secure-by-construction generation, and lifecycle integration. As a structured narrative review, this article does not claim new experimental results; its contribution is a synthesis and design-oriented framework for trustworthy LLM-assisted software security.

Keywords: Large language models; software vulnerability detection; secure code generation; automated program repair; static analysis; software security; code intelligence.

Publication Date : 20.05.2026

1. INTRODUCTION

Software vulnerabilities remain a persistent source of operational, financial, and societal risk. Modern applications combine third-party libraries, distributed services, generated code, and rapidly changing dependencies. Security defects can arise from unsafe input handling, injection flaws, memory errors, insecure authentication, improper authorization, cryptographic misuse, and incorrect resource management. Traditional static application security testing (SAST), dynamic testing, fuzzing, and manual code review each provide valuable evidence, but they can produce false positives, require specialist configuration, or struggle to explain findings in the context of developer intent.

LLMs have introduced a different interaction model. Rather than relying only on predefined rules or learned token patterns, an LLM can summarize a function, relate code to a natural-language requirement, explain a suspected weakness, and draft a candidate repair. Reviews of LLMs in software engineering and software security report growing research activity across code analysis, testing, repair, and security-related tasks [1]–[4]. Yet generative fluency should

not be confused with security assurance: a model can produce a persuasive explanation that is incomplete or incorrect.

This review addresses three questions: (RQ1) how are LLMs being applied to vulnerability detection and secure code generation? (RQ2) what technical and evaluation limitations affect their reliability? (RQ3) how can LLMs be combined with conventional verification tools to support safer development? The paper contributes a task-oriented synthesis, a comparison of approach families, a conceptual hybrid workflow, and a set of research directions.

2. REVIEW METHOD AND SCOPE

This article uses a structured narrative review rather than claiming a fully reproducible systematic review. The literature base was assembled from scholarly publisher and institutional sources, including IEEE Xplore, ACM Digital Library, NIST publications, Frontiers, and proceedings indexed by major research libraries. Search concepts combined "large language model" or "LLM" with "software vulnerability", "secure code generation", "program repair", "static analysis", and "software security". Priority was given to peer-reviewed surveys, empirical studies, benchmark papers, and institutional research published mainly from 2023 through 2026, while foundational datasets and earlier software-security work were retained where relevant.

Studies were considered relevant when they examined at least one of the following: (i) LLM-based vulnerability identification or explanation; (ii) security properties of LLM-generated code; (iii) LLM-supported patch or repair generation; or (iv) evaluation methods and datasets for these tasks. General-purpose code-generation studies without a security dimension were used only when they clarified evaluation limitations. Preprints and technical reports are treated as supplementary evidence, not as equivalent to peer-reviewed journal or conference publications. Because this is a narrative synthesis, no pooled effect size or quantitative meta-analysis is claimed.

The evidence is organized by task and system role rather than by model brand. This avoids treating rapidly changing model versions as stable research categories. Findings are interpreted with attention to task granularity (snippet, function, file, repository), programming-language coverage, vulnerability taxonomy, availability of ground truth, and whether evaluation includes executable tests. These dimensions matter because a model that identifies a suspicious line in a short function is not necessarily capable of detecting a vulnerability whose exploitability depends on repository-wide data flow or runtime configuration.

3. LLMS FOR SOFTWARE VULNERABILITY DETECTION

Vulnerability detection asks whether code contains a security weakness, where it occurs, and why it is exploitable. LLM-based approaches commonly use direct prompting, few-shot examples, retrieval-augmented context, fine-tuning, or agentic tool orchestration. Inputs may include a code snippet, function, file, commit diff, vulnerability description, or surrounding documentation. Outputs range from binary labels to structured findings containing a weakness category, location, rationale, severity estimate, and remediation advice.

A major attraction of LLMs is contextual explanation. They can translate low-level code patterns into developer-oriented reasoning and may connect a code fragment to likely security consequences. However, detection quality is sensitive to the context supplied. Missing call sites,

validation routines, type information, dependency versions, or configuration can lead to both false positives and false negatives. A model may also infer a vulnerability from a familiar pattern even when the code contains a mitigating control.

Recent reviews identify strong growth in LLM vulnerability research but also highlight fragmented datasets, uneven language coverage, inconsistent task definitions, and limited standardization of evaluation [1], [2]. C and C++ receive substantial attention because memory-safety defects are prominent in available datasets, while evidence for other widely used languages and full-stack application contexts remains less uniform. Accordingly, reported performance should not be generalized across languages, weakness classes, or repository scales without direct validation.

4. SECURE CODE GENERATION AND VULNERABILITY REPAIR

Secure code generation differs from vulnerability detection. The objective is not merely to produce code that compiles or passes functional tests; generated code must also satisfy security requirements and preserve intended behavior. A function can return the expected output while remaining vulnerable to injection, authorization bypass, insecure deserialization, path traversal, or unsafe memory access. A systematic review of AI-generated code security emphasizes that code-generation assistance can improve developer productivity while still producing vulnerable implementations [3].

Security-aware prompting can state constraints such as parameterized database access, strict input validation, least privilege, safe error handling, and approved cryptographic APIs. Retrieval-augmented generation can supply organization-specific secure coding standards and trusted examples. Nevertheless, instructions alone do not prove that generated code is secure. The model may omit a constraint, apply it incorrectly, or generate code that is secure in isolation but unsafe when integrated into a larger system.

Automated vulnerability repair adds another layer of difficulty. A repair must remove the weakness while preserving intended functionality, compatibility, performance, and maintainability. NIST research evaluating LLM repair of real-world C/C++ memory-corruption cases found that simpler, localized issues were more tractable than complex defects requiring reasoning across broader program semantics [5]. This distinction suggests that repair systems should estimate scope and uncertainty before attempting an automatic patch.

LLM-based repair pipelines typically identify a defect, generate one or more candidate patches, and validate them using tests or analysis tools. The central risk is patch plausibility without patch correctness. A candidate may silence a warning while leaving the root cause intact, introduce a new weakness, weaken validation, or break a caller's assumptions. Therefore, a patch should be accepted only after independent verification against security-specific and functional requirements.

5. TAXONOMY OF SECURITY TASKS AND VULNERABILITY CLASSES

The term "LLM-based software security" covers several distinct tasks that should not be collapsed into a single detection score. Detection classifies or localizes a weakness in existing code; explanation translates evidence into a developer-oriented account; secure generation produces new code under security constraints; repair modifies vulnerable code; and validation checks whether a proposed result satisfies functional and security requirements. Each task requires different inputs, output formats, and ground-truth evidence.

Task	Typical input	Expected output	Critical validation
Detection	Source code, diff, repository context	Weakness type, location, evidence	Known labels, static/dynamic confirmation
Explanation	Finding plus surrounding code	Cause, impact, exploit preconditions	Reviewer agreement and evidence traceability
Secure generation	Requirement, API and policy context	New implementation	Functional tests plus security tests
Repair	Vulnerable code and defect evidence	Candidate patch	Regression tests and rescanning
Prioritization	Findings, asset and deployment context	Risk-ranked queue	Analyst validation and risk rationale

Vulnerability taxonomies such as the Common Weakness Enumeration (CWE) provide a shared vocabulary for categorizing weaknesses. Nevertheless, a CWE label alone does not fully specify exploitability. For example, an input-validation weakness may be harmless in one execution path but exploitable in another depending on trust boundaries, sanitization, authorization, and downstream use. Review systems should therefore capture both a weakness category and the evidence that links the code behavior to a security impact.

The task taxonomy also clarifies why benchmark results are not directly interchangeable. A model evaluated on function-level binary classification is not being tested for repository-level localization, secure generation, or patch correctness. Research comparisons should report task definition, code granularity, programming language, CWE coverage, context availability, and validation oracle. This reporting discipline is necessary to make evidence useful to software teams choosing an LLM-assisted security workflow.

5.1 Design Implications

A practical system should maintain separate pipelines for detection and repair. Detection can generate candidate findings for triage, while repair should be activated only after a finding is sufficiently supported. The system should preserve a link between each claim, the relevant code span, the tool evidence, and the proposed change. This separation reduces the risk that a persuasive but unsupported model explanation directly triggers code modification.

6. PROPOSED INTELLIGENT HYBRID FRAMEWORK

Based on the reviewed evidence, this paper proposes a conceptual framework for integrating LLM reasoning with conventional software-security verification. The framework is intended as an architectural synthesis, not as an implemented system or a claim of measured performance. Its objective is to make detection and repair traceable, testable, and appropriately supervised.

6.1 Framework Components

(1) Code and context acquisition: collect the relevant source files, commit diff, dependency metadata, build configuration, and developer requirements. Repository context should be minimized to the information needed for analysis while preserving important call relationships and data flows. Sensitive credentials and personal data should be removed or masked before external model processing.

(2) Context preparation and retrieval: construct a structured representation of the task, identify relevant functions and interfaces, and retrieve approved secure-coding guidance. Retrieval sources should be versioned and attributable. The system should distinguish repository facts from retrieved recommendations so that the model cannot silently treat uncertain context as verified evidence.

(3) LLM-assisted analysis: request structured findings containing weakness category, affected location, triggering conditions, evidence from the code, confidence rationale, and suggested next checks. The model should be instructed to return "insufficient evidence" when context is inadequate rather than inventing a vulnerability. Output schemas support downstream validation and audit.

(4) Independent verification: run suitable SAST rules, dependency checks, data-flow or taint analysis, unit tests, integration tests, and—where feasible—fuzzing or dynamic security tests. The LLM finding is compared with tool evidence. Disagreement is retained for human review rather than resolved by allowing the model to overrule deterministic evidence.

(5) Candidate repair and validation: if a defect is confirmed, the LLM may generate a minimal patch and explain its security rationale. The patch is reviewed for scope, tested against regression tests, rescanned with security tools, and checked for unintended behavior. The original source and patch must remain available for audit and rollback.

(6) Human approval and lifecycle feedback: a developer or security reviewer approves, modifies, or rejects the finding and patch. Decisions, tool versions, prompts, model version, retrieved guidance, test outcomes, and final changes are logged. Approved outcomes can inform future evaluation sets, subject to data governance and licensing constraints.

6.2 Workflow

The operational sequence is: source and context intake → context preparation → LLM analysis → independent tool verification → risk prioritization → optional patch generation → functional and security validation → human approval → audit record. A failed validation returns the candidate to repair or manual remediation; it must not be automatically merged. High-severity findings, uncertain exploitability, and changes to authentication, authorization, cryptography, or trust boundaries should require specialist review.

7. EVALUATION CONSIDERATIONS

A credible evaluation should assess detection, repair, security, and operational cost separately. Accuracy alone is insufficient because vulnerability datasets are often imbalanced and a system can achieve high accuracy by predicting the majority class. Precision, recall, F1-score, false-positive rate, and per-weakness performance should be reported for detection. Localization should be measured at line, function, or file level, with the chosen granularity stated explicitly.

For repair, useful measures include patch acceptance rate, test pass rate, security regression rate, functional regression rate, and the proportion of repairs that address the root cause. Evaluation should include adversarial or mutated examples and should distinguish compile success from functional correctness and security correctness. CWEval, for example, emphasizes outcome-driven evaluation of both functionality and security in LLM-generated code [7]. SECODEPLT expands benchmark construction with validated examples and dynamic evaluation artifacts across multiple languages and weakness categories [6].

Comparisons must control for prompt templates, model versions, context windows, sampling settings, tool configurations, and available repository context. Each result should identify whether the model was given vulnerability labels, patches, or test outcomes. Data leakage is a serious concern when benchmark examples overlap with model training data or public repositories. Time-based splits, repository-level separation, and held-out vulnerability families can reduce—but not eliminate—this risk.

Table 2. Recommended Evaluation Dimensions

Dimension	Example measures	Evaluation question
Detection	Precision, recall, F1, false-positive rate	Are real weaknesses found without excessive noise?
Localization	Line/function/file localization accuracy	Can developers locate the reported defect?
Repair	Patch acceptance, root-cause resolution	Does the patch remove the weakness?
Regression	Functional and security regression rate	Does the fix preserve required behavior?
Robustness	Performance across languages and CWE classes	Does performance generalize beyond familiar examples?
Efficiency	Latency, token use, tool runtime, review effort	Is the workflow practical for development teams?
Explainability	Evidence completeness, reviewer agreement	Can findings be independently understood?

Evaluation should also report human effort. A system that finds more candidate weaknesses but requires extensive manual triage may not improve the overall security process. Conversely, an LLM that produces fewer but better-localized findings may provide practical value even when raw recall is not the highest. These trade-offs should be evaluated in realistic developer workflows.

8. CHALLENGES, RISKS, AND RESEARCH GAPS

Context limitations remain a fundamental challenge. Many vulnerabilities depend on interactions across functions, modules, frameworks, deployment settings, and external services. Prompting an LLM with a small code fragment can remove the very information needed to assess exploitability. Repository-level context construction, program slicing, call-graph analysis, and taint tracking may help, but they increase computational and engineering cost.

Hallucination and calibration are also unresolved. LLMs may invent APIs, cite nonexistent code paths, or express unjustified confidence. Confidence scores generated by the model should not be treated as calibrated probabilities unless empirically validated. Structured evidence requirements, abstention mechanisms, and independent tool checks can reduce risk but cannot guarantee correctness.

Benchmark quality and reproducibility are major research gaps. Existing studies use different datasets, programming languages, vulnerability labels, input granularity, prompts, and success criteria. Reviews report that custom and synthetic datasets are common and that repair and explanation tasks are less consistently evaluated than detection [1], [2]. Benchmark contamination and model training overlap further complicate interpretation. Shared artifacts, versioned datasets, executable test oracles, and transparent evaluation protocols are needed.

Secure code generation introduces risks beyond the code itself. Prompts and retrieved documents can contain malicious instructions; source code may include secrets; model providers may process proprietary material; and generated patches may introduce licensing or supply-chain concerns. Organizations should apply data minimization, access controls, approved model policies, dependency pinning, and human review. LLM output must be treated as untrusted input.

9. DISCUSSION AND FUTURE RESEARCH

The reviewed evidence supports a shift from model-centric claims toward workflow-level assurance. Rather than asking whether an LLM can detect vulnerabilities in general, research should specify which weakness classes, languages, repository sizes, and operating conditions are supported. Detection, explanation, and repair should be evaluated as distinct tasks, even when integrated into one pipeline.

Future work should prioritize: (i) multilingual and framework-aware benchmarks; (ii) repository-level context methods with measurable coverage; (iii) calibrated abstention and uncertainty reporting; (iv) secure-by-construction generation constrained by verified APIs and policies; (v) patch validation that combines tests, static analysis, and dynamic evidence; (vi) transparent human-in-the-loop studies; and (vii) privacy-preserving deployment for proprietary repositories. More attention is also needed for underrepresented languages and vulnerability categories.

A further opportunity is continuous security feedback across the software development lifecycle. Findings from code review, CI pipelines, dependency updates, and production incidents can inform a governed knowledge base. Such feedback must be curated: incorrect labels or unreviewed generated patches could otherwise reinforce unsafe patterns. The framework proposed here therefore treats traceability, provenance, and independent verification as core design requirements rather than optional features.

10. LIMITATIONS

This review has limitations. It is a structured narrative synthesis and does not claim exhaustive retrieval of every publication, formal dual-reviewer screening, or a statistical meta-analysis. The field changes rapidly, and model capabilities, APIs, benchmarks, and deployment practices may evolve after the literature considered here. Studies differ substantially in task design and evaluation, limiting direct comparison. Some benchmark papers and institutional reports provide valuable technical evidence but should not be interpreted as proof of broad real-world effectiveness. The proposed framework is conceptual and requires implementation and empirical validation across languages, repositories, and development settings.

11. CONCLUSION

LLMs offer useful capabilities for explaining code, identifying potential weaknesses, generating secure-code candidates, and assisting vulnerability repair. However, their outputs remain probabilistic and can be incomplete, misleading, or insecure. The literature indicates that performance depends on context, programming language, vulnerability type, dataset quality, and evaluation design. Consequently, LLM-only security workflows are not sufficient for high-assurance software development.

This review proposes a hybrid architecture in which LLM reasoning is combined with retrieved security guidance, static and dynamic analysis, executable tests, patch validation, and human approval. Such integration can make LLM assistance more traceable and operationally useful while preserving independent verification. Progress in this area will depend on reproducible benchmarks, broader language coverage, realistic repository-level evaluation, calibrated uncertainty, and responsible data governance. The central conclusion is that LLMs should augment—not replace—established software-security engineering controls.

REFERENCES

1. X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," ACM Transactions on Software Engineering and Methodology, vol. 34, no. 5, Art. 145, 2025. doi: 10.1145/3708522.
2. A Systematic Review on Detection, Repair, and Explanation of Vulnerabilities in Source Code Using Large Language Models," IEEE Access, vol. 13, pp. 192263–192293, 2025. doi: 10.1109/ACCESS.2025.3631363.
3. Kasarla, S. R. (2025). A test case prioritization framework for efficient regression testing in large-scale software systems. *International Journal of Applied Mathematics*, *38*(2s), 1648-1661.
4. A. Negri-Ribalta, A. Geraud-Stewart, and G. Lenzini, "A systematic literature review on the impact of AI models on the security of code generation," Frontiers in Data Science, vol. 7, 2024. doi: 10.3389/fdata.2024.1386720.
5. When Software Security Meets Large Language Models: A Survey," IEEE/CAA Journal of Automatica Sinica, available online Jan. 2025. doi: 10.1109/JAS.2024.124971.
6. L. Zhang, Q. Zou, A. Singhal, X. Sun, and P. Liu, "Evaluating Large Language Models for Real World Vulnerability Repair in C/C++ Code," National Institute of Standards and Technology, June 19, 2024.

7. Y. Nie et al., “SECODEPLT: A Unified Benchmark for Evaluating the Security Risks and Capabilities of Code GenAI,” Advances in Neural Information Processing Systems 38, Datasets and Benchmarks Track, 2025. doi: 10.52202/085713-0447.
8. J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, “CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation,” in Proc. 2nd IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), 2025.
9. F. Zubair, M. Al-Hitmi, and C. Catal, “The use of large language models for program repair,” Computer Standards & Interfaces, vol. 93, Art. 103951, 2025. doi: 10.1016/j.csi.2024.103951.
10. Large Language Models for Software Engineering: A Systematic Literature Review,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 8, Art. 220, 2024. doi: 10.1145/3695988.
11. S. Kaniewski, F. Schmidt, M. Enzweiler, M. Menth, and T. Heer, “A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models,” arXiv:2507.22659, 2025. (Preprint; supplementary evidence.)
12. Kolukuri, N. R. (2025). Beyond dashboards: Designing decision-centric analytics systems that transform organizational decision-making.
13. Liu, B. (2026). State-grounded conversational agency in LLM workflow systems: A human-centered framework for trust, control, and accountable workflow orchestration. Journal of Intelligent Decision Making and Information Science, 3(13s), 411-437.
14. S. R. Kasarla, “An Empirical Analysis of Software Maintainability Using Code Quality and Complexity Metrics,” *Journal of Information Systems Engineering and Management*, vol. 10, no. 4, pp. 2923–2932, 2025. [Online]. Available: https://jisem-journal.com/index.php/journal/article/view/15365