SPECTRUM OF ENGINEERING SCIENCES
ISSN (E) 3007-3138 (P) 3007-312X VOLUME 4, ISSUE 3, 2026

AI-ASSISTED AUTOMATED SOFTWARE TESTING AND VULNERABILITY DETECTION USING LARGE LANGUAGE MODELS

Mustafa Mehboob[*1], Zohaib Muhammad[2], Amna Nazir[3], Hifza Umer[4]

[*1]*Department of Computer Science, Government College University, Lahore, 54000, Pakistan.*
[2]*Department of Computer Science, Government College University, Faisalabad 38040, Pakistan*
[3]*Department of computer science, Fedral urdu University of Art , science and technology Islamabad*
[4]*Department of computer science, FAST NUCES Islamabad*

[*1]mustafaw3451@gmail.com, [2]muhammadzohaibmz4@gmail.com, [3]bintnazir70@gmail.com, [4]hifza.umer@isb.nu.edu.pk

DOI: https://doi.org/10.5281/zenodo.23208092

Keywords
Large Language Models, Automated Software Testing, Vulnerability Detection, Fuzzing, Automated Vulnerability Repair, Agentic AI, Static Analysis, Cybersecurity, Software Quality Assurance, Benchmarking.

Article History
Received: 01 January 2026
Accepted: 16 March 2026
Published: 31 March 2026

Copyright @Author
Corresponding Author: *
Mustafa Mehboob

Abstract

The integration of Large Language Models (LLMs) into software engineering has fundamentally disrupted traditional paradigms of software quality assurance and cybersecurity. This review systematically synthesizes recent empirical advances in LLM-assisted automated software testing and vulnerability detection, evaluating both the transformative capabilities and critical limitations of these emerging technologies. The analysis examines four principal domains: LLM-powered test case generation and fuzzing, static vulnerability discovery, automated vulnerability repair (AVR), and agentic patching frameworks. Empirical evidence demonstrates that LLM-based test generators, including ChatUniTest, CoverUp, and TestPilot, achieve superior branch coverage and mutation scores compared to search-based tools like EvoSuite, with frameworks such as HITS improving coverage from 68.0% to 82.3%. In vulnerability detection, hybrid neuro-symbolic workflows combining traditional SAST tools with LLM verification filters reduce false-positive rates by up to 93.3%, while specialized models like VulnScout-C demonstrate balanced precision-recall trade-offs on dual-verified datasets. Autonomous agentic frameworks, including SWE-agent and OpenHands, achieve 72-74% issue resolution rates on SWE-bench Verified, with the SAN2PATCH framework attaining 79.5% patch success rates. However, this review identifies a critical security correctness disconnect: despite high functional pass rates, AI agents achieve only 10.5-12.5% security correctness on rigorous evaluations, frequently reintroducing insecure legacy functions to satisfy functional tests. Furthermore, agentic workflows exhibit susceptibility to indirect prompt injection and flow-control vulnerabilities, with attack success rates reaching 40.7%. Data contamination in public benchmarks artificially inflates performance metrics, compromising evaluation integrity. The review concludes that while LLMs offer unprecedented semantic reasoning capabilities, their probabilistic nature necessitates robust verification pipelines, hybrid architectures, and security-aware training paradigms to ensure trustworthy deployment across the software development life cycle.

INTRODUCTION

The paradigm of software quality assurance and cybersecurity is experiencing an unprecedented disruption driven by Large Language Models (LLMs). Historically, automated software testing and static vulnerability detection relied on search-based heuristics, symbolic execution, and rigid static analysis rules (alkawi & Alhajj, 2026). While these traditional methodologies established foundational software safety nets, they frequently stumbled when confronted with scale, semantic context, complex object states, and high false-positive overhead. The recent transition toward transformer-based architectures and generative AI has reframed software artifacts source code, test suites, protocol specifications, and security patches as continuous probabilistic tokens requiring high-level semantic reasoning (Rashid et al., 2026).

Modern software engineering frameworks increasingly integrate LLMs across the entire Software Development Life Cycle (SDLC). In automated software testing, LLMs demonstrate exceptional competence in synthesizing contextualized unit tests, generating edge-case payloads for fuzzing engines, and optimizing coverage through execution feedback loops (Han et al., 2025). Concurrently, in software security, LLMs are deployed to perform static analysis, filter static application security testing (SAST) false positives, and drive end-to-end Automated Vulnerability Repair (AVR). Autonomous agentic frameworks such as SWE-agent and OpenHands further extend these capabilities by navigating full software repositories to localize and patch complex vulnerabilities (Saher, 2026).

However, this rapid integration introduces a critical technological paradox. While LLMs excel at generating code that passes functional test cases, their probabilistic nature can lead to subtle semantic failures, data contamination during evaluation, and the introduction of insecure code patterns (Desai et al., 2025). The objective of this review article is to systematically synthesize recent empirical advances in LLM-assisted software testing and vulnerability management. This paper evaluates unit test generation, fuzzing innovations, comparative static analysis performance, agentic repair paradigms, benchmarking suites, and critical security vulnerabilities inherent to AI-driven software development (Al Amin, 2025).

2. LLM-Powered Automated Test Case Generation And Fuzzing

2.1 Unit Test Synthesis and Coverage Optimization

Automated unit test generation aims to minimize manual engineering efforts while maximizing codebase reliability. Search-Based Software Testing (SBST) tools, epitomized by platforms like EvoSuite, historically dominated this domain by utilizing genetic algorithms to optimize structural code coverage. However, SBST tools often produce unreadable, non-idiomatic test cases with low fault-detection capabilities (mutation scores) due to their lack of semantic context regarding the class or method under test (Chu et al., 2025).

The emergence of LLM-based frameworks such as ChatUniTest, CoverUp, ChatTester, and TestPilot has fundamentally shifted test synthesis from pure constraint satisfaction to context-aware program synthesis. Frameworks like ChatUniTest utilize adaptive context-selection mechanisms combined with a generate-verify-repair execution loop (Wang et al., 2024).

The operational mechanism of modern LLM-based test generation relies on an iterative, closed-loop verification pipeline. Initially, an adaptive context selector gathers surrounding source code, abstract syntax trees (ASTs), or relevant documentation via Retrieval-Augmented Generation (RAG) to build a structured prompt. The LLM receives this contextual prompt and synthesizes a candidate test suite (Zhang et al., 2025). This candidate code is immediately dispatched to an isolated execution sandbox where dynamic compilation and test execution take place. If compilation errors or assertion failures occur, the feedback mechanism captures raw stack traces and error metrics, piping them incrementally back into the LLM context. This allows the model to continuously refine its assertions and fix syntax flaws across successive execution iterations (Yang et al., 2026).

Empirical evaluations across benchmark suites such as Defects4J demonstrate that LLM-driven test generators achieve line and branch coverage

comparable to, and in several contexts superior to, search-based tools. For instance, the HITS approach which combines method slicing with LLM prompting boosted code coverage from EvoSuite's 68.0% baseline to 82.3% on target benchmark datasets (Bazalii & Fleischer, 2025). Furthermore, LLM-generated tests consistently achieve higher mutation scores than SBST baselines. This performance differential stems from the LLM's ability to infer method semantics and synthesize meaningful test assertions rather than superficial execution triggers (Jiang, 2026).

Despite these coverage gains, compilation and runtime error rates remain a key operational bottleneck. Evaluation studies reveal that default RAG pipelines and baseline LLMs produce compilation or execution errors in up to 76.2% and 60.9% of generated tests, respectively. Advanced prompting and structural context frameworks reduce these error rates substantially; for example, the APT framework lowered compilation error rates to 25.9%, outperforming both baseline ChatUniTest (60.9%) and EvoSuite (59.6%) (Yao & Liu, 2026).

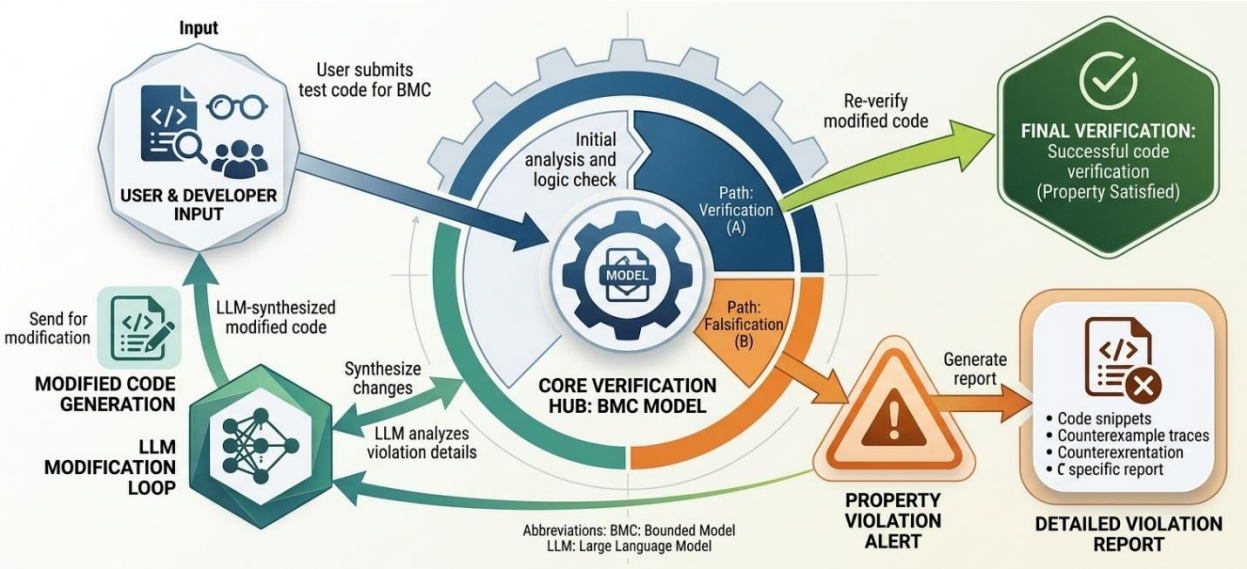


Figure 1: Iterative Verification and Repair Workflow in LLM-Assisted Automated Bounded Model Checking

2.2 Generative and Edge-Case Fuzzing

Fuzzing is an automated testing technique that feeds malformed or unexpected inputs into a target program to uncover abnormal behaviors, memory corruption, and unhandled exceptions. Traditional fuzzers rely on random byte mutation or static grammar definitions, which struggle when target programs require highly structured inputs or deep multi-variable dependencies (Deng et al., 2023).

LLMs have transformed generative fuzzing by acting as intelligent mutation engines and input format generators. In deep learning library testing (e.g., PyTorch, TensorFlow), tools such as TitanFuzz, FuzzGPT, and Magneto leverage LLMs to synthesize complex, edge-case tensor operations and API call chains (Deng et al., 2024). FuzzGPT demonstrates that priming LLMs with historical bug-triggering code snippets enables the model to hallucinate unusual, valid syntax variants that bypass preliminary validation checks, successfully triggering hidden library edge cases. Similarly, the ∇Fuzz architecture integrates coverage-guided feedback directly into prompt selection, outperforming conventional deep learning library fuzzers in both line coverage and zero-day bug discovery (Hu et al., 2023).

In network protocol and stateful application testing, tools such as ChatAFL leverage LLMs to automatically infer message grammars from protocol documentation or raw message logs. By querying the model to generate grammar-

compliant network packets, the fuzzer penetrates deeper state spaces without requiring manual rule construction (Shi et al., 2026).

Table 1: Performance Benchmarks of LLM Testing Frameworks vs. Traditional SBST Tools

Tool / Framework	Underlying Methodology	Target Domain / Language	Key Performance Metrics & Findings	Primary Operational Advantage
EvoSuite	Search-Based Software Testing (SBST / Genetic Algorithms)	Java	Baseline coverage ~68.0%; high initial pass rate, variable mutation score	Deterministic execution without model hallucination costs
ChatUniTest	LLM + Dynamic Compiler Verification Loop	Java / Python	Superior branch coverage & higher mutation scores than EvoSuite; initial error rate ~60.9%	Context-aware assertion logic and functional test readability
CoverUp	Coverage-Guided Iterative Prompting	Python	High regression test coverage by targeting uncovered lines iteratively	Dynamic context injection based on coverage analysis
FuzzGPT	LLM Priming with Historical Bug Payloads	C++ / Python (DL Libraries)	Synthesizes unusual edge-case code; uncovers novel zero-day library crashes	Primed mutation generation targeting deep architectural layers
∇Fuzz	Neural Coverage-Guided Dynamic Fuzzing	Python (DL Frameworks)	Substantially outperforms state-of-the-art DL fuzzers in coverage and bug hits	Gradient-inspired prompt adjustment via real-time code coverage
ChatAFL	LLM-Guided Protocol Grammar Synthesis	C / Stateful Network Protocols	Dramatically higher state coverage compared to AFL-style protocol fuzzers	Automated extraction and generation of complex protocol message grammars

3. Vulnerability Detection: LLMs vs Traditional Static Analysis (SAST)

3.1 Static Vulnerability Discovery and Semantic Reasoning

Static Application Security Testing (SAST) tools including CodeQL, Semgrep, SonarQube, and Snyk Code rely on abstract syntax trees (ASTs), control flow graphs (CFGs), and dataflow taint analysis to detect security flaws. While SAST engines excel at deterministic pattern matching for known vulnerability types, they suffer from two major operational constraints: high false-positive rates that overburden security teams, and an inability to reason about cross-file business logic flaws (Sheng et al., 2025).

Generative LLMs, particularly reasoning-focused architectures (e.g., DeepSeek-R1, GPT-4, Claude 3.5 Sonnet), treat vulnerability detection as a deep semantic reasoning task. By processing code semantics holistically, LLMs evaluate context, developer intent, and complex variable interactions that evade rigid AST matching rules (Hossain & Shapna, 2025). Comparative empirical benchmarks reveal stark differences between traditional SAST tools and modern code-aware LLMs. On real-world CVE detection datasets (such as CWE-Bench), advanced LLMs

and AI security agents significantly outperform standard static analyzers in detection rates. For example, in standardized comparative evaluations, the specialized AI security tool Trent achieved a detection rate of 0.250, followed by Claude Code at 0.087, whereas traditional SAST tools such as CodeQL and Semgrep achieved detection rates of only 0.037 and 0.036, respectively (Iranmanesh et al., 2025).

However, raw detection capability in baseline LLMs is frequently counterbalanced by lower precision and unpredictable recall across diverse CWE categories. Models fine-tuned on dedicated vulnerability datasets (e.g., VulnScout-C, PrimeVul) demonstrate improved trade-offs between precision and recall by restricting hallucination tendencies through dual-verified training samples (Gnieciak & Szandala, 2025).

3.2 Hybrid SAST-LLM Synergies and False Positive Filtering

Rather than replacing traditional SAST, the most effective security deployment paradigm uses hybrid neuro-symbolic workflows. In these hybrid frameworks, static analyzers provide fast, rule-grounded candidate flags, while LLMs act as intelligent verification filters to eliminate false positives (Agrawal & Ahi, 2025).

The operational architecture of a hybrid SAST-LLM discovery framework executes in sequential stages. Raw repository source code is initially ingested by a deterministic static analysis engine, such as CodeQL or Semgrep, which scans for known pattern signatures and taint paths. Because these tools generate high volumes of candidate alerts containing substantial false-positive noise, the raw alert outputs are redirected into an LLM-based verification engine (Agrawal & Ahi, 2026). Frameworks like IRIS leverage semantic context and execution flow reasoning to evaluate whether candidate alerts constitute true vulnerabilities or false alarms. Verified security defects are then passed downstream to automated remediation pipelines for agentic patching (Pavlyshyn & Krasko, 2026).

Empirical data highlights the efficacy of this synergy. The IRIS system, which integrates CodeQL dataflow analysis with LLM context evaluation, reduced the baseline false-positive rate of CodeQL by 5% in Java vulnerability scans while simultaneously discovering true positives missed by standard queries. On real-world enterprise Java codebases, specialized LLM agent configurations achieved a false-positive identification rate of up to 93.3% when reviewing raw CodeQL alerts (De Rosa, 2026).

Furthermore, LLMs are increasingly used upstream to generate and refine complex static analysis rules. By converting high-level natural language vulnerability descriptions or historic CVE patches into executable CodeQL queries, LLMs eliminate the steep learning curve associated with authoring custom static analysis rules (Thornton, 2026).

Table 2: Comparative Metric Profile of Static Analysis Tools and LLM Architectures

System / Tool Class	Underlying Mechanism	Primary Strengths	Precision / Recall Characteristics	Vulnerability Discovery Vulnerabilities & Limitations
Traditional SAST (e.g., CodeQL, Semgrep)	AST / CFG Taint Analysis & Dataflow Rules	High speed, deterministic, zero hallucination; catches known syntactic patterns	High recall on standard patterns; prone to high False Positive (FP) rates	Struggles with cross-file business logic, custom sanitizers, and dynamic features
General Reasoning LLMs (e.g., GPT-4, DeepSeek-R1)	Deep Semantic Attention & Generative Inference	Analyzes developer intent, contextual flow, and complex logic flaws	High contextual sensitivity; variable precision depending on prompt structure	Susceptible to hallucinations and non-deterministic false positives

Specialized Small Models (e.g., VulnScout-C)	Fine-Tuned Compact Transformer Architecture	Highly optimized resource consumption; rapid line-level C/C++ detection	High precision and balanced F1 score on dual-verified vulnerability sets	Narrower domain adaptability beyond specific target language subsets
Hybrid SAST + LLM (e.g., IRIS Framework)	Rule-Based Filtering + LLM Context Verification	Eliminates false positive noise while preserving static query rigor	Filters up to 93.3% of SAST false positives; achieves optimal F1 score balance	Overhead of multi-stage execution and LLM token consumption costs

4. Automated Vulnerability Repair (AVR) and Agentic Patching

4.1 Patch Synthesis and Validation Architectures

Automated Vulnerability Repair (AVR) extends vulnerability detection to the automatic synthesis of secure, functional code patches. Early zero-shot prompting techniques applied to AVR often resulted in plausible-looking patches that introduced subtle secondary security flaws, broke compilation, or disrupted adjacent business logic (Yang et al., 2025).

To address these shortcomings, modern AVR architectures employ structured reasoning paradigms combined with dynamic validation environments. Advanced reasoning strategies such as Tree-of-Thought (ToT) prompting, Monte Carlo Tree Search (APRMCTS), and Generate-Verify-Repair loops explore multiple patch candidate trajectories simultaneously (Seo et al., 2025). For example, the SAN2PATCH framework utilizes a Tree-of-Thought LLM analysis pipeline to evaluate program state transformations before committing a patch, achieving a 79.5% patch success rate (31 out of 39 verified vulnerabilities fixed) on benchmark evaluation suites (Kim et al., 2026).

In these iterative repair frameworks, synthesized patches must satisfy a multi-tier verification pipeline:

- **Syntactic and Structural Validation**: The patch must compile cleanly without introducing syntax errors or build failures.
- **Functional Regression Testing**: The modified program must pass all existing unit and integration test suites to confirm that core system utility remains intact.
- **Security Proof-of-Concept (PoC) Validation**: The patch must fail to trigger the original security exploit script or dynamic sanitizer alert (e.g., AddressSanitizer), confirming vulnerability mitigation (Liu et al., 2026).

4.2 Autonomous Software Engineering Agents in Security

The evolution of LLMs from simple prompt-response interfaces to autonomous software engineering agents has fundamentally expanded the scope of AVR. Frameworks such as SWE-agent, OpenHands (formerly OpenDevin), Trae Agent, Aider, and VulnAgent-X operate directly inside full software repositories using interactive shell tools, execution sandboxes, and file-system search utilities (Chhabra et al., 2026).

Autonomous agentic repair pipelines operate through a multi-step execution loop within an interactive sandbox. Upon receiving a bug report or dynamic proof-of-concept (PoC) exploit script, the core agentic coordinator such as SWE-agent or OpenHands traverses the codebase using repository search tools and AST analysis to localize the root defect (Vakhula & Opirskyy, 2025). The agent then constructs candidate patches, which are executed dynamically inside the sandbox environment against both existing regression test suites and the specific PoC exploit script. If regression tests pass and the PoC exploit is neutralized, the agent commits the secure patch directly to the repository and drafts a pull request for human audit (Otoum & Elkhalili, 2026).

Agentic evaluation highlights critical architectural trade-offs based on tool-task fit. SWE-agent excels at dispersed syntactic searches requiring iterative file traversal across large directory trees.

Conversely, OpenHands performs better when vulnerabilities require centralized patch reasoning and deep architectural modifications within a localized module (Moradian & Håkansson, 22010). On the SWE-bench Verified benchmark, OpenHands achieves a 72% successful issue resolution rate, while streamlined implementations like Mini-SWE-Agent reach 74% using compact execution prompts. On security-focused subsets targeting real-world CVEs, SWE-agent achieves a 33.8% successful patch rate and a 12.5% PoC resolve rate on complex multi-file instances (Karuppuchamy, 2026).

5. Benchmarking Frameworks, Security Risks, and Emerging Pitfalls

5.1 Benchmark Landscape

Rigorous benchmarking is essential for evaluating the comparative efficacy of AI-driven testing and vulnerability tools. Benchmark datasets have evolved from synthetic, small-scale function snippets to complex, repository-level multi-language environments (Abbasi et al., 2025).

Table 3: Overview of Key Benchmarks for LLM Vulnerability Detection and Repair

Benchmark Suite	Target Domain & Focus Areas	Dataset Scale & Primary Composition	Key Evaluation Metrics	Primary Evaluation Value
Big-Vul	Historical C/C++ Vulnerability Detection	~11,800 C/C++ vulnerabilities mapped to CVEs	Precision, Recall, F1 Score	Standard baseline for functional vulnerability detection
PrimeVul	High-Quality Realistic Vulnerability Detection	Chronologically split, dual-verified C/C++ CVEs	Data-contamination-resistant F1 Score	Reduces label noise and prevents temporal data leakage
SWE-bench / SWE-bench Verified	Repository-Level Software Engineering & Repair	Real-world GitHub issues across major Python projects	Issue Resolve Rate, Pass@1, Patch Success	Evaluates autonomous agent capabilities in live repositories
CyberSecEval (1–4)	LLM Cybersecurity Risks & Defense Capabilities	Standardized prompt injection, secure code generation, & exploit suites	Pass Rate, Security Correctness Rate, Attack Success Rate	Measures offensive abuse risks and defense generation quality
SEC-bench	Real-World Security Tasks for LLM Agents	Real-world CVE patching and PoC generation tasks	Patch Success Rate (>18%), PoC Resolve Rate (>10%)	Directly evaluates autonomous agent security resolution
NYU CTF Bench	End-to-End Offensive Cybersecurity Capabilities	Capture-The-Flag (CTF) vulnerability exploitation challenges	Challenge Completion Rate	Tests deep multi-step exploitation and analysis capabilities

5.2 Security Correctness Disconnect, Data Contamination, and Vulnerabilities

Despite high benchmark scores in standard code completion tasks, recent research reveals a severe disconnect between functional pass rates and actual security correctness. Assessments by the Agent Security League demonstrated that when AI agents are evaluated on security correctness (ensuring code not only works, but is immune to common security weaknesses like buffer overflows or injection flaws), performance drops drastically (Luo et al., 2018). On evaluations combining SWE-agent with Claude Sonnet 4, the system achieved only a 10.5% security correctness rate,

while the highest scoring framework (OpenHands) reached just 12.5%. Models consistently prioritize functional code completion over non-functional security constraints, often re-introducing insecure legacy functions (e.g., strcpy, unauthenticated SQL strings) to make functional test cases pass (Al-Maamari, 2026).

Furthermore, AI agents operating autonomously within software environments introduce significant security attack vectors. Agentic workflows running in untrusted environments are highly susceptible to Indirect Prompt Injection and Flow-Control Vulnerabilities (FCVs). If an agent inspects an untrusted repository containing a malicious string hidden inside a comment or source file, the injection payload can hijack the agent's context, forcing it to execute arbitrary shell commands or exfiltrate private API keys. Studies evaluating OpenHands paired with GPT-5 Mini demonstrated a 40.7% attack success rate when exposed to FCV threat vectors (Aslan et al., 2023). Finally, evaluation integrity is heavily compromised by data contamination. Because models are trained on public GitHub repositories, standard benchmarks (e.g., Big-Vul, early SWE-bench instances) frequently leak into LLM pre-training corpora, resulting in artificially inflated pass rates that degrade when deployed on un-indexed enterprise codebases (Akhundov, R., & Ibrahimov, 2025).

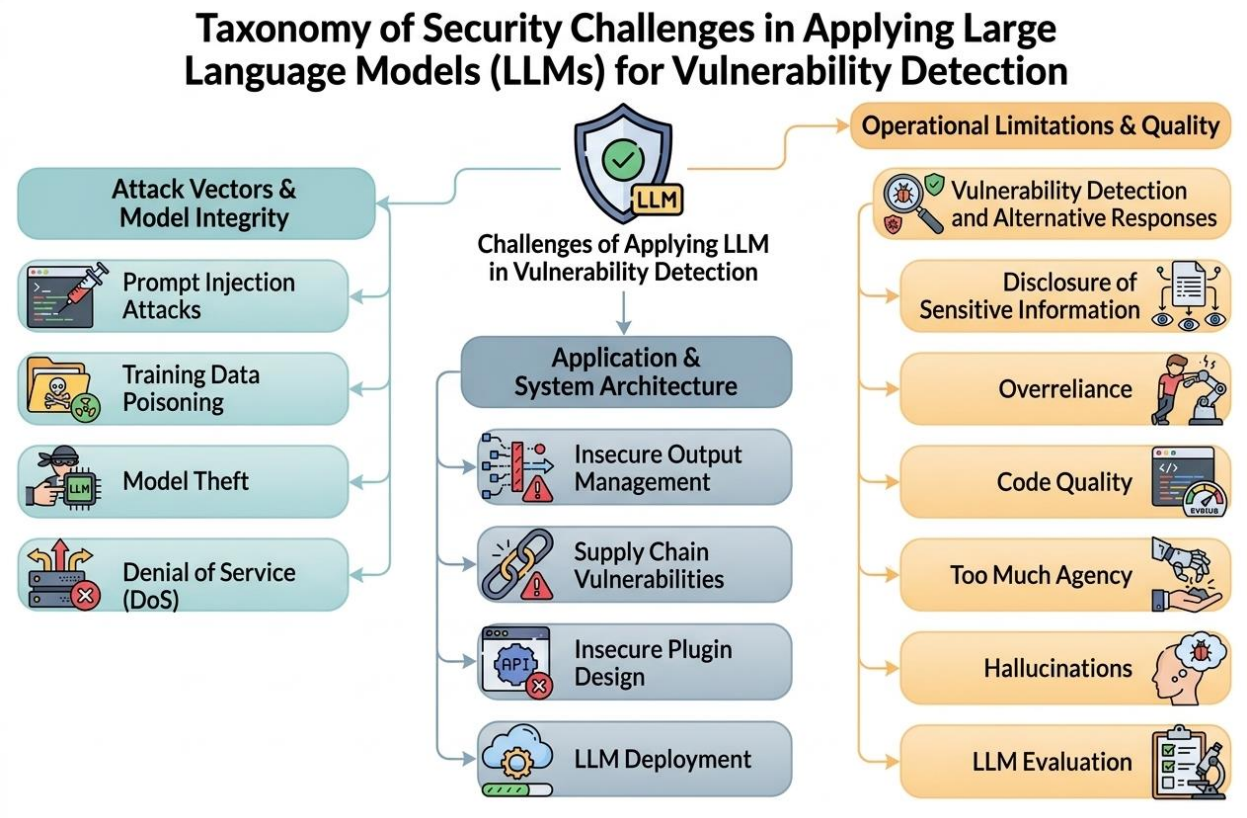


Figure 2: Taxonomy of Security Challenges, Attack Vectors, and Operational Limitations in LLM-Based Vulnerability Detection

Conclusion

This review has systematically examined the rapidly evolving landscape of LLM-assisted software testing and vulnerability detection, revealing a technological paradigm shift from deterministic, rule-based methodologies toward probabilistic, context-aware semantic reasoning. The empirical evidence synthesized across four

principal domains test case generation, fuzzing, static vulnerability analysis, and automated repair demonstrates that LLMs deliver substantial improvements in coverage, mutation detection, and contextual vulnerability discovery. Hybrid neuro-symbolic architectures, particularly those coupling traditional SAST tools with LLM verification engines, have proven especially effective, achieving false-positive reduction rates exceeding 93% while preserving deterministic rigor. Autonomous agentic frameworks have expanded the scope of automated repair, enabling repository-level patch synthesis with success rates approaching 80% on benchmark evaluations. However, this review also identifies critical challenges that temper unqualified optimism. The security correctness disconnect where agents achieve functional success rates above 70% but security correctness rates below 13% represents a fundamental limitation requiring urgent attention. The susceptibility of agentic workflows to indirect prompt injection and flow-control vulnerabilities, with attack success rates exceeding 40%, poses significant deployment risks in untrusted environments. Additionally, data contamination in public benchmarks undermines evaluation integrity, artificially inflating reported performance metrics. Future research must prioritize security-aware training objectives, robust multi-tier verification pipelines, and contamination-resistant benchmark suites. The path forward lies not in replacing traditional static analysis but in forging synergistic hybrid architectures that combine the deterministic precision of rule-based tools with the semantic flexibility of generative models. Only through such balanced integration, coupled with rigorous security validation at every stage, can LLMs fulfill their promise as trustworthy collaborators in software quality assurance and cybersecurity defense.

REFERENCES

Abbasi, M., Cardoso, F., Váz, P., Silva, J., & Martins, P. (2025). A practical performance benchmark of post-quantum cryptography across heterogeneous computing environments. *Cryptography*, *9*(2), 32.

Agrawal, V., & Ahi, K. (2025). Llm-driven sast-genius: A hybrid static analysis framework for comprehensive and actionable security. *arXiv preprint arXiv:2509.15433*.

Agrawal, V., & Ahi, K. (2026). Reducing Noise: Hybrid Static Application Security Testing–Large Language Model Pipeline for Code Security. *IEEE Security & Privacy*, *24*(3), 59-64.

Akhundov, R., & Ibrahimov, B. (2025). Ensuring Information Security within the Military Environmental Safety System. *Моделювання, керування та інформаційні технології*, (8), 178-183.

Al Amin, S. (2025). AI or LLM assisted software testing: a mapping study.

Al-Maamari, A. (2026). Why LLMs Fail: A Failure Analysis and Partial Success Measurement for Automated Security Patch Generation. *arXiv preprint arXiv:2603.10072*.

Aslan, Ö., Aktuğ, S. S., Ozkan-Okay, M., Yilmaz, A. A., & Akin, E. (2023). A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. *Electronics*, *12*(6), 1333.

Bazalii, M., & Fleischer, M. (2025). Orion: Fuzzing Workflow Automation. *arXiv preprint arXiv:2509.15195*.

Chhabra, A., Datta, S., Nahin, S. K., & Mohapatra, P. (2026). Agentic AI security: Threats, defenses, evaluation, and open challenges. *IEEE Access*.

Chu, B., Feng, Y., Liu, K., Guo, Z., Zhang, Y., Shi, H., ... & Xu, B. (2025). Large language models for unit test generation: Achievements, challenges, and opportunities. *arXiv preprint arXiv:2511.21382*.

De Rosa, R. (2026). *A Hybrid Multi-Agent Architecture for Enhancing CodeQL Static Analysis with Large Language Models* (Doctoral dissertation, Politecnico di Torino).

Deng, Y., Xia, C. S., Yang, C., Zhang, S. D., Yang, S., & Zhang, L. (2023). Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014*.

Deng, Y., Xia, C. S., Yang, C., Zhang, S. D., Yang, S., & Zhang, L. (2024, February). Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering* (pp. 1-13).

Desai, P., Singh, S., & Amilkanthwar, S. (2025). The Test Pyramid 2.0: AI-assisted testing across the pyramid. *Frontiers in Artificial Intelligence*, *8*, 1695965.

Gnieciak, D., & Szandala, T. (2025). Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. *IEEE Access*, *13*, 198410-198422.

Han, S., Kim, H., Lee, H., Moon, H., Jeon, Y., Bae, H., ... & Lee, S. (2025, November). Dependable code repair with llms: Ai-driven vulnerability detection and automated patching. In *2025 IEEE 30th Pacific Rim International Symposium on Dependable Computing (PRDC)* (pp. 176-181). IEEE.

Hossain, S. M. S., & Shapna, A. M. (2025). Modern approaches to software vulnerability detection: A survey of machine learning, deep learning, and large language models. *Electronics*, *14*(22), 4449.

Hu, J., Zhang, Q., & Yin, H. (2023). Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*.

Iranmanesh, M., Sabet, S. M., Marefat, S., Ghasr, A. J., Wilson, A., Sharafaldin, I., & Tayebi, M. A. (2025). Zerofalse: Improving precision in static analysis with llms. *arXiv preprint arXiv:2510.02534*.

Jiang, Y. (2026). *Toward Bidirectional Integration of Neural Networks and Software Testing: From Enhancing Fuzzing to Validating ROS-based Autonomous Systems* (Doctoral dissertation, Macquarie University).

Karuppuchamy, S. (2026, February). Governed Agentic AI for Software Platforms: A Reference Architecture for Safe Autonomy at Scale. In *SoutheastCon 2026* (pp. 1-6). IEEE.

Kim, W., Min, S., Gwon, M., Baik, D., Lee, H., Heo, H., ... & Yun, I. (2026). PatchIsland: Orchestration of LLM Agents for Continuous Vulnerability Repair. *arXiv preprint arXiv:2601.17471*.

Liu, S., Liu, F., Wang, P., Li, T., Zhu, Y., Lian, X., & Zhang, L. (2026). ContraFix: Skill-Enhanced Contrastive Runtime Analysis for Vulnerability Repair. *arXiv preprint arXiv:2605.17450*.

Luo, L., Zhang, Y., Pearson, B., Ling, Z., Yu, H., & Fu, X. (2018). On the security and data integrity of low-cost sensor networks for air quality monitoring. *Sensors*, *18*(12), 4451.

Malkawi, M., & Alhajj, R. (2026). AI-powered vulnerability detection and patch management in cybersecurity: A systematic review of techniques, challenges, and emerging trends. *Machine Learning and Knowledge Extraction*, *8*(1), 19.

Moradian, E., & Håkansson, A. (2010, September). Controlling security of software development with multi-agent system. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems* (pp. 98-107). Berlin, Heidelberg: Springer Berlin Heidelberg.

Otoum, N., & Elkhalili, N. (2026). Methods and techniques of agentic software engineering: A systematic literature review. *IEEE Access*, *14*, 7443-7465.

Pavlyshyn, V., & Krasko, O. (2026). SAST IMPROVEMENTS USING LLM FOR CI/CD PIPELINES.

Rashid, M. B., Hossain, M. S. J., Khan, M. I., Tahora, S., Siddika, A., Prakash, M. I., ... & Shahriar, H. (2026). A Survey on Large Language Models in Software Security: Opportunities and Threats. *Computers*, *15*(4), 226.

Saher, Y. (2026). AI-Assisted Code Review and Quality Assurance: A Comparative Analysis of Code Smell and Security Vulnerability Detection.

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

Sheng, Z., Chen, Z., Gu, S., Huang, H., Gu, G., & Huang, J. (2025). Llms in software security: A survey of vulnerability detection techniques and insights. *ACM Computing Surveys*, *58*(5), 1-35.

Shi, J., Yin, X., Huang, J., Tao, S., & Zhao, J. (2026, July). Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 2352-2382).

Thornton, S. (2026). Can Adversarial Code Comments Fool AI Security Reviewers-Large-Scale Empirical Study of Comment-Based Attacks and Defenses Against LLM Code Analysis. *arXiv preprint arXiv:2602.16741*.

Vakhula, O., & Opirskyy, I. R. (2025). AI-driven Security as Code for software development using multi-agent systems. In *CH&CMiGIN* (pp. 170-185).

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, *50*(4), 911-936.

Yang, B., Cai, Z., Liu, F., Le, B., Zhang, L., Bissyandé, T. F., ... & Tian, H. (2025). A survey of LLM-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*.

Yang, Z., Inani, K., Kabra, K., Gupta, V., & Iyer, A. P. (2026). SAFuzz: Semantic-Guided Adaptive Fuzzing for LLM-Generated Code. *arXiv preprint arXiv:2602.11209*.

Yao, Z., & Liu, S. (2026, July). Failure Modes of Standalone LLM-Based Test Generation. In *2026 20th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)* (pp. 505-510). International Institute of Applied Informatics.

Zhang, Q., Fang, C., Gu, S., Shang, Y., Chen, Z., & Xiao, L. (2025). Large language models for unit testing: A systematic literature review. *arXiv preprint arXiv:2506.15227*.

IEER
Excellence in Education & Research