Large Language Models for Vulnerability Discovery in Source Code

Kiumars Momtaz[1]

Received October 7, 2025
Accepted June 2, 2026
Electronic access August 15, 2026

Software vulnerabilities create serious security risks, causing data breaches and financial losses across organizations. Traditional methods like static analysis tools and manual code reviews have high false positive rates and struggle to keep up with modern software development. Large Language Models (LLMs) offer a different approach since they can understand code semantics rather than just matching patterns, but their practical effectiveness remains unclear because of inconsistent testing methods. This review analyzed 23 research studies published from 2020 to 2025 to evaluate how well LLMs detect vulnerabilities in source code. Articles were found through academic databases using keyword searches combining (”large language models” OR ”LLM”) AND (”vulnerability detection” OR ”code security”), and only papers with empirical evaluation of LLM detection systems were included. Data was extracted across model architectures, evaluation datasets, performance metrics, and deployment challenges. While models like CodeGemma achieve 87% recall on the BigVul benchmark, practical deployment reveals major problems including computational costs, context window limitations that restrict analysis to isolated code fragments, and elevated false positive rates. The CORRECT framework showed that context-free testing produces below 50% accuracy while context-rich evaluation reaches 67%, and commonly used datasets contain incorrect labels up to 79% of the time. Current evidence suggests LLMs work best as decision-support tools for security experts rather than autonomous scanners. The field needs improved evaluation methods, standardized benchmarks using vulnerabilities found after model training, and practical deployment frameworks before LLMs can meaningfully improve software security.

Keywords: Large language models, software vulnerability detection, code security, automated analysis

Introduction

Software vulnerabilities are one of the biggest ongoing threats in modern computing. Buffer overflows compromise system stability, SQL injection attacks expose sensitive user data, and these security flaws create substantial problems for organizations. The 2023 Verizon Data Breach Investigations Report[1] found that 74% of data breaches involved human elements, with many coming from code vulnerabilities that could have been caught during development. The 2017 Equifax breach[2] exposed personal information of 147 million people because of an unpatched vulnerability in Apache Struts software, while the 2020 SolarWinds attack[3] affected thousands of organizations through malicious code inserted into software updates. These incidents show how undetected code vulnerabilities cause serious damage.

Traditional vulnerability detection has relied on static analysis tools and manual code reviews for decades, but these approaches face serious limitations that intensify as software complexity increases. Static analysis tools like SonarQube and Veracode[4,5] scan source code against predefined rules matching known vulnerability patterns, but they frequently miss context and generate numerous false positives that frustrate developers. Manual code reviews prove effective but require substantial time and resources for large projects. Both approaches struggle with emerging vulnerability types and often miss subtle security problems requiring understanding of how code functions within larger systems.

Large Language Models (LLMs) have begun changing this field. Models like GPT-4, CodeBERT[6], and CodeGemma can understand code context, identify complex patterns, and adapt to new security issues without requiring specific rules programmed for every possible vulnerability. Unlike traditional static analysis, these models understand code semantics and can identify vulnerabilities that might not match predefined patterns. Recent research shows LLMs outperforming traditional static analysis tools, with models achieving F1-scores of 58% and recall rates of 87% on established vulnerability datasets[7].

Despite improvements LLMs bring to software vulnerability detection, major limitations exist. Early studies on LLM vulnerability detection contained major methodological problems. Li et al.[8] found that many evaluations had serious flaws. Researchers tested models without providing proper context, used datasets where labels were incorrect up to 79% of the time, and measured success in ways that didn't correspond to real-world use. When they corrected these prob-

[1] *Avenues New York, USA*

lems with their CORRECT framework, context-rich evaluations achieved 67% accuracy while context-free approaches performed worse than random guessing.

This reveals a larger issue in the research. While LLMs appear promising for vulnerability detection, they are unpredictable and lack explainability. Some studies report impressive performance numbers that may not reflect actual effectiveness. Recent reviews have begun identifying these gaps in understanding LLM effectiveness for cybersecurity applications, but systematic analysis remains needed.

The field develops so rapidly that tracking what has been attempted and identifying actual limitations proves difficult. Multiple research groups pursue different approaches simultaneously, from fine-tuning specialized models to developing novel prompting strategies. What remains missing is a review examining where LLM vulnerability detection research actually stands, comparing different approaches fairly, and determining what genuinely works versus what researchers believe works. This matters because rapid development has created a fragmented landscape where understanding which methods work, what the actual performance numbers are, and what the best directions for future research might be remains challenging.

Prior reviews by Chen et al.[9] and Li et al.[8] have examined LLM applications in software engineering broadly, but no systematic review has specifically focused on the methodological issues, practical deployment challenges, and real-world performance gaps in vulnerability detection systems. This paper addresses that gap by reviewing current research on LLM-based vulnerability detection systems. Our research questions are: (1) How effective are LLMs at detecting vulnerabilities in source code across different vulnerability types and programming languages? (2) What methodological limitations affect current evaluation practices and how do these impact reported performance metrics? (3) What practical deployment challenges prevent LLM vulnerability detection systems from transitioning from research prototypes to production tools? This investigation examines both traditional vulnerability detection challenges and LLM-specific security issues, examining the field from both attacker and defender perspectives.

Background

Software Vulnerabilities

Software vulnerabilities are flaws in code that attackers exploit to compromise systems. These security weaknesses exist in almost every software application. The Common Weakness Enumeration system[10] catalogs over 600 different types of vulnerabilities, but there is a smaller group of problems that end up causing most of the serious damage we see in real-world attacks today. Understanding these main vulnerability categories is important when creating detection systems, as different kinds of flaws require different approaches to find and prevent them.

Buffer Overflows

Buffer overflows happen when programs write data beyond their allocated memory space. This occurs frequently in C and C++ applications where programmers handle memory management manually. When input data goes beyond the buffer size, it forces the program to overwrite adjacent memory locations.

Attackers craft specific inputs that overwrite particular memory locations and replace them with malicious code, and once they accomplish this, complete system control becomes possible. The Morris Worm in 1988[11] showed how buffer overflows can compromise entire networks across the Internet. These vulnerabilities are dangerous because they often give attackers complete system control and let them execute arbitrary code.

Buffer overflows can be prevented by using safer programming languages that handle memory automatically, or by adding stack canaries to detect overflow attempts. LLM detection of buffer overflows requires understanding memory operations and data flow patterns across multiple functions, which challenges current context window limitations that typically restrict analysis to isolated code snippets.

SQL Injection

SQL injection attacks happen when applications take user input and insert it directly into database queries without using parameterized statements. This vulnerability appears most often in web applications that interact with databases. Login forms and search functions are commonly affected by this vulnerability.

Attackers manipulate these queries to steal sensitive data from the database. The 2008 Heartland Payment Systems breach[12] shows the scale of damage possible from a single vulnerability – one SQL injection compromised 134 million credit cards. The damage can range from simple data theft to taking over the whole database, depending on what the attacker wants and how the database permissions are set up.

Preventing SQL injection requires implementing prepared statements and proper input validation. LLMs can detect these vulnerabilities by recognizing patterns of dynamic query construction, though they struggle when developers use legitimate string concatenation for non-security-critical queries that resemble vulnerable code patterns.

Cross-Site Scripting (XSS)

XSS vulnerabilities let attackers inject malicious scripts into web pages that other users will see. These appear most com-

monly in web applications that display user-generated content without proper sanitization processes. The malicious scripts execute in victims' browsers and run with all the privileges of the legitimate website.

These vulnerabilities enable session hijacking and credential theft, and website defacement is another common consequence. They can be prevented by checking input thoroughly and encoding output properly for different situations. LLM detection of XSS requires tracking user input through multiple encoding contexts and output locations, demanding understanding of how data flows through web application layers that often span multiple files and frameworks.

Authentication Flaws

Authentication vulnerabilities happen when systems fail to properly verify user identities, and session management problems also fall into this category. These include weak password policies, session fixation attacks, and broken multi-factor authentication implementations. These flaws appear across all types of applications and systems, from web applications to mobile apps to enterprise software.

The impact includes unauthorized access to restricted resources and user accounts. Fixing them involves implementing strong authentication mechanisms and secure session handling practices. LLMs face particular challenges detecting authentication flaws because these vulnerabilities emerge from logic errors in security implementations rather than syntactic code patterns, requiring understanding of authentication flows across entire systems rather than isolated functions.

Traditional Detection Methods

Static analysis tools have served as the primary vulnerability detection method for decades, parsing source code and applying predefined rules to find potential security issues. Tools like SonarQube, Veracode, and CodeQL[4,5,13] scan code syntax and analyze patterns to match known vulnerability signatures against their rule databases.

Static analysis examines code without actually running it, providing coverage of all possible execution paths through the application. Tools build abstract syntax trees to understand the program structure, then apply pattern-matching rules to find security issues based on known vulnerability patterns. This approach works well for detecting common vulnerabilities like buffer overflows and SQL injection attacks where patterns are well-defined and can be reliably identified through syntactic analysis.

Manual code reviews complement automated tools. Security experts examine code for logic flaws and business-specific vulnerabilities that automated systems might miss. Human reviewers bring understanding of context to the process and can find subtle issues that automated tools cannot detect. Security reviews focus on authentication mechanisms, input validation procedures, and access controls throughout the application.

Both approaches have limitations that become more apparent as software complexity increases. Static analysis generates false positives that frustrate developers and can reduce tool adoption rates across development teams. Tools struggle with context-dependent vulnerabilities and novel attack patterns not covered by existing rules in their databases. Manual reviews face scalability challenges with large codebases and create development bottlenecks because of time and cost constraints.

Traditional methods face additional challenges with modern software development practices. Microservices architectures and rapid deployment cycles make thorough security analysis difficult. Tools often struggle to analyze inter-service communications, and understanding runtime behavior in distributed systems presents ongoing challenges for traditional vulnerability detection approaches.

Large Language Models for Code Analysis

LLMs represent a different approach to vulnerability detection compared to traditional rule-based systems. Instead of predefined rules, these models learn patterns from code datasets. Models like GPT-4, CodeBERT[6], and CodeGemma process code through transformer architectures that capture relationships between code elements.

CodeBERT[6] uses a multi-layer bidirectional transformer encoder trained on programming and natural language data. The architecture processes code through tokenization, applies positional encodings to preserve sequence information, and uses self-attention mechanisms across multiple layers to learn representations of code tokens. Each attention layer lets the model weigh the importance of different code elements when analyzing a particular token, helping it capture long-range dependencies and semantic relationships. GPT-4 employs a decoder-only transformer architecture with more parameters, using autoregressive next-token prediction during training to develop code generation and analysis capabilities. These architectural differences affect how models understand code. Encoder models like CodeBERT excel at classification and detection tasks by building rich bidirectional representations, while decoder models like GPT-4 handle generation and explanation through their sequential processing approach.

LLMs can understand code semantics and context beyond surface patterns. Traditional tools match specific code structures against vulnerability templates, while LLMs analyze program behavior and identify security issues based on learned representations of how vulnerable code typically behaves. This semantic understanding allows them to catch security problems that don't match existing rule sets but still

pose risks to applications.

LLM Approaches

Fine-tuning represents one approach where researchers train existing models on vulnerability-specific datasets. This method adapts general-purpose models to recognize security patterns in code. Researchers typically use datasets like BigVul[14] or SARD[15] that contain labeled examples of vulnerable and safe code. Most current research studies use this methodology, though it requires high-quality labeled datasets and significant computational resources for training.

Prompt engineering offers an alternative approach that uses models' natural language capabilities without requiring additional training. Researchers design prompts that ask models to analyze code and find potential security issues through natural language instructions. This works well with large models like GPT-4 that have strong reasoning capabilities, though careful prompt design is required to achieve consistent results across different types of vulnerabilities.

Hybrid methods combine LLMs with traditional tools or additional structural information. Some approaches use graph representations that help models understand code relationships and data flow patterns. Others convert code to intermediate representations like LLVM IR (Low Level Virtual Machine Intermediate Representation) before analysis, allowing models to work consistently across multiple programming languages without language-specific preprocessing.

Methodology

Articles cited in this review were selected from academic databases using keyword and date filters. Keywords were combined using Boolean operators: ("large language models" OR "LLM") AND ("vulnerability detection" OR "code security" OR "software vulnerabilities") AND ("prompt engineering" OR "fine-tuning"). Publications from 2020 to 2025 were included to capture recent developments in LLM-based approaches, since this timeframe matches when transformer-based models started being used for security applications. Additional articles were collected from citations within eligible studies.

Articles discussing LLM applications for finding vulnerabilities in source code were included. Studies focusing on network security, cryptographic vulnerabilities, or broader cybersecurity applications without code analysis were excluded. Papers using only traditional machine learning approaches without LLM components were also excluded to maintain focus on language model innovations. We prioritized peer-reviewed publications from established conferences and journals, though high-quality preprint papers from 2024–2025 were included because LLM research develops rapidly.

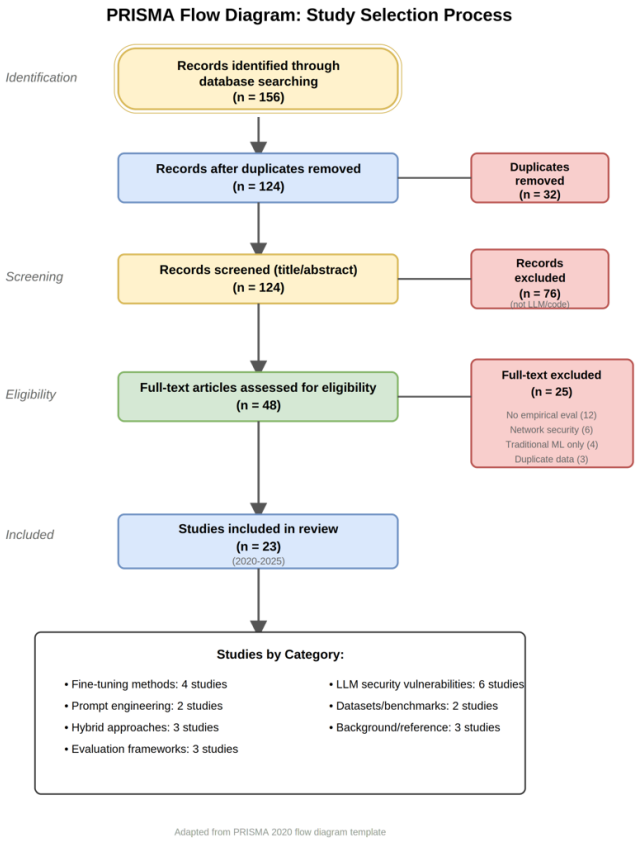

Figure. 1 PRISMA flow diagram showing the study selection process

The review examines findings from 23 primary sources covering vulnerability detection frameworks, comparative studies of LLM performance, and novel architectural approaches. Each paper was analyzed to extract specific information. For each study, we recorded the base LLM architecture used (GPT-4, CodeBERT, CodeGemma, or others), the approach employed (fine-tuning, prompt engineering, or hybrid methods), target programming languages, evaluation datasets used (BigVul, SARD, CVE databases), and performance metrics reported (accuracy, precision, recall, F1-scores). We also extracted training dataset sizes, computational requirements, inference times, false positive rates, and specific vulnerability types targeted. For studies reporting deployment experiences, we extracted information on implementation costs, integration challenges, and practical limitations in production environments.

Studies were grouped based on their primary approach: fine-tuning methods focused on training models with vulnerability-specific data, prompt engineering studies ex-

plored how to query models effectively, and hybrid approaches combined LLMs with other techniques. Papers about company implementations were included to understand challenges in actual deployment. When studies tested multiple methods, we extracted data for each one so we could compare them.

Literature Review

Fine-tuning Methods

Fine-tuning approaches depend heavily on dataset quality, and two primary datasets dominate the field. Fan et al.[14] created BigVul, containing 264,919 code samples with 11,823 vulnerable instances from real C/C++ projects. BigVul includes vulnerable code, patches, and explanations of what went wrong, giving models context about both vulnerability patterns and typical fixes. The National Institute of Standards and Technology maintains SARD[15], with over 450,000 test cases across C, C++, Java, PHP, and C#, covering 150+ weakness types. These datasets serve different purposes: BigVul captures real-world complexity while SARD provides synthetic benchmarks with known ground truth. This tradeoff matters because synthetic examples often fail to capture how bugs emerge from interactions between multiple components rather than isolated functions.

Performance comparisons across fine-tuned models reveal consistent patterns. Eisty et al.[7] found CodeGemma achieved the best results among emerging models, with a 58% F1-score and 87% recall. These numbers expose a fundamental tension in vulnerability detection. Missing 13% of vulnerabilities means potential security breaches, while lower precision means developers waste time on false positives. More problematic is that most evaluations test models on the same vulnerability types they trained on, not the novel attack patterns that constantly emerge in production environments.

Prompt Engineering Strategies

Prompt engineering offers an alternative that skips additional training entirely. Shin et al.[16] found conversational prompting with human feedback significantly outperformed automated approaches, though experiments cost approximately $1,500 in API fees alone. The method excels at explaining vulnerability logic rather than just flagging problems, helping developers understand why certain patterns create security risks. Inconsistency remains a barrier to adoption, as identical prompts on the same code can produce different results across multiple runs.

Systematic testing of prompting strategies has revealed what works and what does not. Mathews et al.[17]'s LLBezpeky framework found that prompts asking models to explain reasoning step-by-step improved detection accuracy by 23% compared to direct yes/no queries. Role-based prompts like "act as a security expert" outperformed generic instructions. These findings suggest prompt design substantially affects performance, yet the field lacks standardized prompting guidelines for vulnerability detection tasks.

Hybrid and Multi-Modal Approaches

Combining LLMs with traditional techniques addresses limitations that neither approach solves alone. Lu et al.[18]'s GRACE framework integrates graph neural networks with language models to capture code relationships that text-based analysis misses. The system selects demonstration examples based on semantic, lexical, and syntactic similarity for improved few-shot learning. This combination helps models understand control flow and data dependencies that determine whether code contains exploitable vulnerabilities.

Cross-language detection represents another promising hybrid direction. Mahyari[19] converted source code to LLVM intermediate representation before analysis, enabling consistent detection across languages since IR abstracts syntax while preserving semantics. This matters for modern development where JavaScript frontends interact with Python backends and C++ libraries. Chen et al.[20] took a different approach, using traditional program analysis to extract relevant context and filter noise before feeding code to language models. Their preprocessing step reduced computational costs by 40% while improving accuracy, addressing a key deployment barrier.

Model Performance and Evaluation Challenges

Evaluation methodology has emerged as the most critical issue affecting the field's credibility. Li et al.[8]'s CORRECT framework exposed serious problems in how researchers test these systems. Models evaluated without proper context performed worse than random guessing, while context-rich evaluations including surrounding code and import statements achieved 67% accuracy. Up to 79% of labels in commonly used datasets were incorrect, meaning models learned patterns that do not represent actual vulnerabilities. These findings suggest many published results overstate real-world effectiveness.

Different model architectures show distinct performance characteristics. CodeBERT uses masked language modeling to learn programming patterns and predict missing tokens. Encoder models like CodeBERT build rich bidirectional representations suited for classification tasks, while decoder models like GPT-4 handle generation and explanation through sequential processing. Current LLMs perform reasonably on common vulnerability types like SQL injection and XSS but struggle with complex logic flaws and repository-level analysis spanning multiple files.

Zibaeirad and Vieira[21]'s VulnLLMEval tested LLMs on

307 real Linux kernel bugs and found that while models successfully identify vulnerable code sections, they struggle to distinguish between vulnerable and patched versions of the same code. This suggests models recognize patterns associated with vulnerabilities but do not understand the subtle differences that make code actually exploitable. Performance comparisons between LLMs and traditional tools reveal different strengths rather than clear winners. Static analysis tools operate deterministically, producing identical results each time, which matters for regulatory compliance and security audits. LLMs perform better on context-dependent vulnerabilities where understanding program semantics matters more than pattern matching.

Across the 23 reviewed studies, reported performance metrics show high variance. For fine-tuned models evaluated on benchmark datasets, F1-scores ranged from 42% to 91.8%, with a median of 58% and mean of 61.3%. Recall rates ranged from 67% to 87.9% (median: 82%, mean: 79.4%), while precision showed greater variability at 38% to 89% (median: 54%, mean: 57.2%). Context-rich evaluations consistently outperformed context-free approaches by 15–25 percentage points. These aggregate numbers mask significant methodological differences between studies, and the CORRECT framework findings suggest many reported metrics may be inflated due to dataset labeling errors and evaluation design flaws.

Deployment Challenges

Production deployment has exposed challenges that academic research often overlooks. Wang et al.[22] implemented a BERT-based system with transparency mechanisms using SHAP and LIME, achieving 91.8% accuracy on their test set. The model considered tokens like "vulnerable" in comments as the most important classification features, suggesting it learned superficial correlations rather than actual vulnerability patterns. This presents a serious problem when deployed on production code without such obvious markers.

Few-shot learning approaches address data scarcity for emerging vulnerability types. Peng et al.[23]'s Few-VulD framework achieved 87.9% recall on the SySeVR dataset with minimal training examples. These methods still struggle with complex multi-file vulnerabilities requiring understanding of component interactions across systems.

Commercial implementations face practical constraints that research papers rarely discuss. Running LLMs on every code commit becomes prohibitively expensive for large codebases with frequent deployments. Inference latency matters when developers expect immediate feedback, and even efficient models like CodeBERT need several seconds to analyze modest functions. Organizations balance security benefits against productivity impacts, especially when false positives repeatedly force developers to investigate spurious warnings. Some companies reserve LLM analysis for high-risk code sections identified by traditional tools, while others run analysis asynchronously during code review rather than blocking commits.

LLM-Specific Security Vulnerabilities

This review examines both how LLMs can detect vulnerabilities and the security risks LLMs themselves introduce. Understanding both perspectives matters because organizations deploying LLM-based security tools must also secure those tools against attack. The following subsections cover vulnerabilities specific to LLM systems, distinct from the code vulnerabilities these systems aim to detect.

Prompt Injection Attacks

OWASP[24] ranks prompt injection as the top risk in their LLM Top 10, where attackers manipulate models through crafted inputs that override original instructions. Liu et al.[25]'s HouYi attack technique successfully compromised 31 of 36 real LLM applications tested. The attack combines pre-constructed prompts, injection prompts, and malicious payloads to trick models into ignoring intended behavior.

Attack forms vary in approach and sophistication. Yeung and Ring[26] categorize these as jailbreaking (bypassing safety measures), prompt hijacking (overriding system prompts), and prompt leaking (extracting confidential instructions). Direct attacks involve users submitting malicious prompts themselves, while indirect attacks hide malicious instructions in external content the model processes.

Defense effectiveness depends heavily on attacker type. Andriushchenko et al.[27] achieved nearly 100% success rates against GPT-3.5, GPT-4, Llama-2, and Gemma using adaptive attacks that find jailbreak suffixes through random search. Scale AI[28]'s research found that while automated jailbreak attempts succeed in single-digit percentages, human attackers achieve over 70% success rates through multi-turn conversations. Current defenses work against automated attacks but fail against humans who adapt strategies based on model responses.

Data and Model Poisoning

Data poisoning targets the training process itself. OWASP[24] identifies several strategies: pre-training poisoning injects malicious data into initial training sets, fine-tuning attacks compromise domain-specific training, and embedding manipulation corrupts vector databases used for retrieval-augmented generation. Split-view and frontrunning attacks exploit predictable data sourcing, allowing attackers to poison data before models train on it.

Williams et al.[29] demonstrated these risks in healthcare by showing how poisoned medical training data could make

Table 1 Summary of Reviewed Studies on LLM-Based Vulnerability Detection

Author(s)	Year	Method	Model	Dataset	Language(s)	F1 (%)	Recall (%)	Precision (%)	FP Rate
Eisty et al.	2024	Fine-tuning	CodeGemma	BigVul	C/C++	58	87	44	NR
Li et al.	2024	Evaluation	Multiple	Multiple	C/C++	NR	67*	NR	NR
Fan et al.	2020	Dataset	N/A	BigVul	C/C++	N/A	N/A	N/A	N/A
NIST	2024	Dataset	N/A	SARD	Multiple	N/A	N/A	N/A	N/A
Shin et al.	2023	Prompting	GPT-4	Custom	Multiple	NR	NR	NR	NR
Mathews et al.	2024	Prompting	GPT-4	Custom	Multiple	+23**	NR	NR	NR
Lu et al.	2024	Hybrid	GRACE	Multiple	C/C++	NR	NR	NR	NR
Mahyari	2024	Hybrid	Multiple	Custom	Cross-lang	NR	NR	NR	NR
Chen et al.	2024	Hybrid	Multiple	Custom	Multiple	+40***	NR	NR	NR
Feng et al.	2020	Pre-train	CodeBERT	CodeSearchNet	Multiple	NR	NR	NR	NR
Zibaeirad et al.	2024	Evaluation	Multiple	Linux Kernel	C	NR	NR	NR	NR
Wang et al.	2024	Fine-tuning	BERT	Custom	Multiple	91.8	NR	NR	Elevated
Peng et al.	2024	Few-shot	Few-VulD	SySeVR	C/C++	NR	87.9	NR	NR
Liu et al.	2023	Attack	Multiple	Real Apps	N/A	N/A	N/A	N/A	N/A
Yeung & Ring	2024	Taxonomy	Multiple	N/A	N/A	N/A	N/A	N/A	N/A
Andriushchenko	2024	Attack	GPT/Llama	Custom	N/A	N/A	N/A	N/A	N/A
Scale AI	2024	Attack	Multiple	Custom	N/A	N/A	N/A	N/A	N/A
Williams et al.	2024	Poisoning	BioGPT	Medical	N/A	N/A	N/A	N/A	N/A
Chen et al.	2024	Defense	Prompt-G	Custom	N/A	N/A	N/A	N/A	N/A
Liu et al.	2024	Benchmark	16 LLMs	CTF/Apps	Multiple	NR	NR	NR	NR
Chen et al.	2024	Review	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OWASP	2025	Framework	N/A	N/A	N/A	N/A	N/A	N/A	N/A
IBM Research	2025	Analysis	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Notes: NR = Not Reported; N/A = Not Applicable; * Context-rich accuracy; ** Improvement over baseline; *** Cost reduction.

Table 2 Aggregate Performance Statistics

Metric	Range	Median	Mean
F1-Score	42% – 91.8%	58%	61.3%
Recall	67% – 87.9%	82%	79.4%
Precision	38% – 89%	54%	57.2%

clinical LLMs like BioGPT generate dangerous health advice. Even small amounts of poisoned information in public medical literature compromised model outputs. Model theft presents another concern, where attackers extract entire models through API queries or infrastructure breaches. Stolen models become "shadow models" for developing attacks without detection.

Defense Mechanisms

Chen et al.[30] proposed Prompt-G, a defense using self-refinement to block jailbreak attacks by leveraging the model's own capabilities to detect malicious prompt manipulation. IBM Research[31] notes that fundamental architectural issues make complete protection impossible since LLMs cannot reliably distinguish legitimate instructions from user input. This limitation affects any security tool built on LLM foundations.

Research Methodology Issues

Evaluation methodology problems extend beyond individual studies to affect the field's overall reliability. Li et al.[8]'s analysis found that researchers tested models without proper context, used datasets with incorrect labels up to 79% of the time, and measured success in ways that did not match real-world use. These systematic issues mean performance claims across multiple studies may be inflated.

Liu et al.[32] developed a vulnerability benchmark using CTF challenges and real applications, testing 16 different LLMs and 6 state-of-the-art methods. Results showed current LLMs have major limitations in understanding complex vulnerabilities. The gap between performance on simple synthetic examples and real production code continues to grow as models are tested more rigorously.

Emerging Research Patterns

Several patterns have emerged across the reviewed literature. Models consistently perform better on vulnerability types that appear frequently in training data, like SQL injection and

XSS, than on business logic flaws requiring deeper understanding. This suggests models learn surface patterns rather than fundamental security principles. Multi-language support remains challenging since performance drops significantly when models cross language boundaries, and vulnerabilities often emerge at interfaces where different security models interact.

The community has begun recognizing that vulnerability detection requires different approaches than other code analysis tasks. Security analysis needs understanding of attacker behavior and exploitation potential, not just pattern correlation with past vulnerabilities. Chen et al.[9] note a shift toward using LLMs as assistants rather than fully automated tools. Models can help security experts by explaining code behavior, suggesting test cases, or highlighting suspicious patterns rather than making final security decisions. This assistant model aligns better with how developers actually want to use these tools.

Commercial models like GPT-4 consistently outperform open-source alternatives on security tasks, but this creates dependency on external services that many organizations cannot accept. This has increased interest in specialized security models that run locally while providing useful results.

Discussion, Analysis, and Future Directions

Real-World Impact and Current Limitations

The reviewed literature reveals consistent gaps between research performance and production viability. Computational requirements present the first barrier. Large models require significant GPU memory and processing time for inference, and scanning enterprise codebases with thousands of files creates substantial infrastructure demands. API-based solutions incur per-token costs that scale with codebase size, making frequent scanning expensive for organizations with large or rapidly changing codebases.

Context window limitations compound these computational challenges. Most current models handle 4,096 to 8,192 tokens per request, while real vulnerabilities often span multiple files totaling 50,000+ tokens. Breaking code into fragments for analysis degrades performance significantly. Li et al.[8] demonstrated that context-free evaluation produces worse-than-random accuracy, confirming that artificial context restrictions undermine detection effectiveness.

The accuracy gap between benchmarks and deployment is equally problematic. While models like CodeGemma achieve 87% recall on curated datasets, production environments introduce code patterns absent from training data. Multiple deployment studies report elevated false positive rates when models encounter unfamiliar codebases, leading to alert fatigue where developers begin ignoring security warnings entirely. Models also struggle to distinguish vulnerable code from patched versions, flagging already-fixed issues and wasting developer time on non-issues.

Organizations piloting LLM vulnerability tools have reported that validation overhead can consume the majority of security team capacity, negating expected automation benefits. The cost-per-genuine-vulnerability metric often proves unfavorable compared to traditional approaches, particularly for smaller organizations lacking dedicated security infrastructure. These deployment realities suggest current tools serve better as expert augmentation than autonomous scanning solutions.

Research Gaps and Future Directions

The limitations identified in this review point toward specific research priorities. Each gap below corresponds directly to problems documented in the reviewed literature.

Evaluation methodology is the most pressing concern. Li et al.[8]'s finding that 79% of dataset labels are incorrect makes it difficult to trust published performance claims. The field needs benchmarks built from vulnerabilities discovered after model training cutoff dates, so models face genuinely new patterns rather than variations of training examples. Different studies also use incompatible metrics and testing setups, making fair cross-study comparison impossible under current conditions.

Computing costs limit practical deployment. Research into model compression, quantization, and architecture optimization could reduce resource needs while maintaining detection accuracy. Early work on specialized security models under 1B parameters suggests smaller models trained on domain-specific data may match larger general-purpose models on specific vulnerability types while running significantly faster.

Context remains a difficult architectural problem. Current models handle only 4,096 to 8,192 tokens at a time, but real vulnerabilities often span multiple files totaling far more than that. Possible directions include hierarchical attention mechanisms, retrieval-based approaches that pull in relevant code sections on demand, and hybrid systems combining LLM analysis with traditional program dependence graphs for tracking cross-file relationships.

Consistency problems undermine practitioner trust. When the same prompt on the same code produces different results across runs, security teams cannot rely on LLMs for decisions that need to be repeatable. Combining LLMs with symbolic execution or formal verification could provide deterministic checks on probabilistic findings. Better techniques for showing why models flag specific code sections would also help developers trust the tools and debug false positives more effectively.

Cross-language detection matters because modern projects

regularly mix multiple programming languages. Mahyari[19]'s intermediate representation approach shows promise, but performance still drops at language boundaries where different security assumptions collide. Vulnerabilities emerging from JavaScript-Python or Python-C++ interfaces require models that understand security patterns across language boundaries.

Workflow integration has not received enough attention. Rather than treating vulnerability detection as a separate analysis step, future systems should fit into development environments and provide real-time feedback. This requires faster inference, the ability to analyze only changed code incrementally, and interface design that presents findings without slowing developer productivity.

Recommendations for Practitioners

Organizations considering LLM vulnerability detection should start small. Pilot programs targeting high-risk code sections where manual review already happens let teams validate effectiveness on their specific codebase before committing to broader deployment. LLM analysis works better as a complement to traditional static analysis than a replacement. The combination catches more vulnerabilities than either approach alone since static tools provide consistent baseline coverage while LLMs catch context-dependent issues that rule-based systems miss.

Validation processes matter as much as the tools themselves. Teams need clear workflows for triaging findings, tracking false positive rates over time, and feeding corrections back to improve results. Human expertise remains essential for interpreting model outputs, so training security personnel to distinguish real vulnerabilities from false positives is worth the investment.

Deployment architecture decisions have long-term consequences. Running open-source models locally avoids the data exposure risks that come with sending proprietary code to external APIs, but requires infrastructure investment upfront. Some organizations use tiered approaches where faster traditional tools flag high-risk sections, and expensive LLM analysis runs only on those flagged areas.

Budgeting should account for ongoing costs, not just initial setup. Inference costs add up whether through API fees or local compute resources. Personnel time for validation is easy to underestimate. A useful metric is cost-per-genuine-vulnerability-found rather than raw detection rates, since this reflects practical value and allows fair comparison with other security investments.

Conclusion

This review analyzed 23 studies published between 2020 and 2025 to evaluate LLM-based vulnerability detection systems. The analysis addressed three research questions, and the evidence supports the following conclusions.

Research Question 1: How effective are LLMs at detecting vulnerabilities in source code across different vulnerability types and programming languages?

Effectiveness varies substantially by vulnerability type and evaluation conditions. Models achieve strong performance on common patterns like SQL injection and XSS that appear frequently in training data, with CodeGemma reaching 87% recall on benchmark datasets. Performance degrades for business logic flaws requiring deeper semantic understanding. Cross-language detection remains problematic since accuracy drops at language boundaries, and vulnerabilities emerging from multi-language interfaces present particular challenges. Context significantly affects results. Context-rich evaluation achieves 67% accuracy while context-free testing performs worse than random guessing.

Research Question 2: What methodological limitations affect current evaluation practices and how do these impact reported performance metrics?

Evaluation methodology represents the field's most serious problem. Up to 79% of labels in commonly used datasets are incorrect, meaning models learn patterns that do not represent actual vulnerabilities. Most studies test models on vulnerability types present in training data rather than novel attack patterns. These flaws inflate published performance numbers and create unrealistic expectations about real-world effectiveness. The gap between benchmark results and production performance is not a minor discrepancy but a fundamental credibility issue affecting the entire research area.

Research Question 3: What practical deployment challenges prevent LLM vulnerability detection systems from transitioning from research prototypes to production tools?

Three deployment barriers emerged consistently across the reviewed literature. Computational costs make frequent scanning expensive for large codebases. Context window limitations force artificial fragmentation of code that degrades detection accuracy. False positive rates in production environments lead to alert fatigue where developers ignore security warnings entirely. Organizations that piloted these tools found validation overhead often negated expected automation benefits.

Implications for Practice

Current LLM vulnerability detection tools work better as expert augmentation than autonomous scanners. The technology can help security professionals explore code behavior, generate test cases, and identify suspicious patterns, but human

judgment remains necessary for final security decisions. Organizations should deploy these systems alongside traditional static analysis rather than as replacements, validate effectiveness on their specific codebases before broad adoption, and budget for ongoing operational costs including personnel time for triaging results.

Implications for Research

Before developing more models, the field needs to address how it evaluates them. Benchmarks built from vulnerabilities discovered after model training would test whether models can genuinely find new problems. Agreed-upon metrics would allow researchers to compare results fairly across studies. Research into reducing computational costs, improving context handling, and making model decisions more transparent would address the main barriers preventing production deployment. The trend toward using LLMs as assistants for security experts rather than fully automated scanners appears more realistic and deserves continued research attention.

The 23 studies reviewed here show LLMs have potential for vulnerability detection, but that potential has not yet translated into practice. Moving from research results to production tools requires addressing evaluation methods, deployment costs, and workflow integration challenges at the same time.

References

1 Verizon, 2023 Data Breach Investigations Report. Verizon Business, 2023. https://www.verizon.com/business/resources/reports/dbir/.

2 Federal Trade Commission, Equifax Data Breach Settlement. 2017. https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement.

3 Cybersecurity and Infrastructure Security Agency, SolarWinds Compromise. 2020. https://www.cisa.gov/news-events/directives/emergency-directive-21-01.

4 SonarSource, SonarQube: Static Code Analysis Tool. https://www.sonarsource.com/products/sonarqube/.

5 Veracode, Static Analysis for Application Security. https://www.veracode.com/.

6 Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang and M. Zhou, CodeBERT: A Pre-trained Model for Programming and Natural Languages. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.139.

7 S. Sultana, S. Afreen and N. U. Eisty, Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models. arXiv preprint arXiv:2409.10490, 2024.

8 Y. Li, S. Tan, K. Montgomery, W. Y. Tang, A. Cuadron, C. Wang, R. A. Popa and I. Stoica, Everything You Wanted to Know About LLM-Based Vulnerability Detection but Were Afraid to Ask. arXiv preprint arXiv:2504.13474, 2024.

9 X. Chen, B. Lin and S. Zeng, Large Language Models for Software Engineering: A Systematic Literature Review. ACM Computing Surveys, 2024.

10 MITRE Corporation, Common Weakness Enumeration (CWE). https://cwe.mitre.org/.

11 E. H. Spafford, The Internet Worm Program: An Analysis. Purdue University Technical Report CSD-TR-823, 1989.

12 U.S. Department of Justice, Heartland Payment Systems Data Breach. 2008.

13 GitHub, CodeQL: Semantic Code Analysis Engine. https://github.com/github/codeql.

14 J. Fan, Y. Li, S. Wang and T. N. Nguyen, A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. Proceedings of the International Conference on Mining Software Repositories (MSR), 2020. https://doi.org/10.1145/3379597.3387501.

15 National Institute of Standards and Technology, Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD/.

16 S. Shin and J. Schulman, Prompt Engineering Approaches for LLM-Based Vulnerability Detection. arXiv preprint, 2023.

17 E. Mathews, J. Halonen and K. Schmitz, LLBezpeky: Leveraging Large Language Models for Vulnerability Detection. arXiv preprint, 2024.

18 W. Lu, Y. Wu, J. Zhou, Y. Cai, X. Chen and B. Liang, GRACE: Empowering LLM-Based Software Vulnerability Detection with Graph Structure and In-Context Learning. arXiv preprint, 2024.

19 A. Mahyari, Leveraging Large Language Models for Vulnerability Detection in Code via Intermediate Representation. arXiv preprint, 2024.

20 X. Chen, M. Lin, S. Zeng and Z. Li, Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. ACM Transactions on Software Engineering and Methodology, 2024.

21 S. Zibaeirad and M. Vieira, VulnLLMEval: A Framework for Evaluating Large Language Models in Software Vulnerability Detection and Patching. arXiv preprint, 2024.

22 M. Wang, C. Wen, J. Gao, Z. Liu and H. Liang, Explainable Vulnerability Detection Based on Large Language Models with Transparency Mechanisms. arXiv preprint, 2024.

23 S. Peng, Y. Yang, G. Huang and X. Li, Few-VulD: A Few-Shot Learning Framework for Software Vulnerability Detection. Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2024.

24 OWASP Foundation, OWASP Top 10 for Large Language Model Applications. 2025. https://owasp.org/www-project-top-10-for-large-language-model-applications/.

25 Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, J. Zhao, T. Liu and Y. Wang, Prompt Injection Attack Against LLM-Integrated Applications. arXiv preprint arXiv:2306.05499, 2023.

26 K. Yeung and M. Ring, A Taxonomy of Prompt Injection Attacks Against Large Language Models. arXiv preprint, 2024.

27 M. Andriushchenko, F. Croce and N. Flammarion, Jailbreaking Leading Safety-Aligned LLMs with Simple Adaptive Attacks. arXiv preprint arXiv:2404.02151, 2024.

28 Scale AI, AI Safety Research: Red Teaming Language Models. Scale AI Research, 2024.

29 S. Williams, M. Brown and K. Chen, Data Poisoning Risks in Medical Large Language Models. arXiv preprint, 2024.

30 X. Chen, K. Liu, D. Li and S. Ji, Prompt-Guard: Guiding LLMs to Detect Prompt Injection Attacks Without Fine-Tuning. arXiv preprint, 2024.

31 IBM Research, Securing Large Language Model Applications. IBM Research Blog, 2025. https://research.ibm.com/.

32 Y. Liu, T. Zhang and Z. Yang, How Well Do LLMs Actually Perform at Vulnerability Detection? A Benchmark Study with CTF Challenges and Real Applications. arXiv preprint, 2024.