Leveraging Large Language Models for Security-Focused Code Reviews

Author: Dan McQuade, dtmcquade@gmail.com

Advisor: Fletus Poston

Accepted: February 23rd, 2025

Abstract

This study investigates the potential application of Large Language Models (LLMs) in enhancing software security through automated vulnerability detection during the code review process. The research examines the efficacy of LLMs in identifying security vulnerabilities that human reviewers, particularly those without extensive security backgrounds, might overlook. Through analysis of historically significant Common Vulnerabilities and Exposures (CVEs) in popular open-source projects, including frameworks such as Django and Log4j, this research evaluates the capability of LLMs to detect subtle security flaws within complex codebases. The methodology employs a phased approach to LLM prompting, progressing from general code analysis to targeted vulnerability identification while maintaining controlled conditions by isolating vulnerable code segments. By comparing LLM performance against traditional human code reviews and automated security scanning tools, this study provides crucial insights into the potential role of artificial intelligence in augmenting software security practices. The findings suggest implications for the evolution of code review methodologies and the integration of AI-assisted security analysis within software development lifecycles.

1. Introduction

In January 2021, a critical vulnerability in SolarWinds' Orion network monitoring software led to one of the most advanced supply chain attacks ever witnessed. The vulnerability, which presumably had passed through multiple automated code reviews unnoticed, allowed attackers to inject malicious code into software updates, ultimately compromising thousands of organizations, including multiple U.S. federal agencies. This breach exemplifies a persistent challenge in software security— even with established code review practices, critical vulnerabilities can escape detection, particularly when reviewers lack specialized security expertise.

Modern software development increasingly relies on code reviews as a fundamental quality control mechanism. However, as Yu et al. (2024) demonstrate through their analysis of vulnerability-containing source files, these reviews face significant limitations in security vulnerability detection. Their research reveals a systematic gap between traditional code review practices and the specialized expertise required for effective security analysis. This gap is particularly evident in the historical vulnerability patterns of widely used open-source projects.

The emergence of advanced Large Language Models (LLMs) presents a potentially transformative approach to this challenge. Recent research by Zhou et al. (2024) has begun to map the theoretical frameworks through which LLMs might augment human code review capabilities, particularly in security contexts. Three models—GitHub Copilot, Google Gemini, and Anthropic's Claude—have demonstrated sophisticated capabilities in code analysis that warrant systematic investigation for security review applications.

This study examines the feasibility and effectiveness of these LLMs as automated security vulnerability detection tools through a focused analysis of three historically significant open-source projects: Django, Log4j, and the Sudo utility. These projects were selected for their diverse technological contexts: Django represents modern web framework vulnerabilities in Python, Log4j exemplifies Java-based logging infrastructure security challenges, and Sudo illustrates system-level security considerations in C. Through this carefully curated cross-section of critical software infrastructure, this

research evaluates the LLMs' capability to identify security flaws across different programming languages and vulnerability classes.

Building on Alrashedy and Aljasser's (2023) work on feedback-driven security patching, this study systematically examines documented Common Vulnerabilities and Exposures (CVEs) within these projects. The research methodology draws on Yin and Ni's (2024) frameworks for evaluating LLM performance in vulnerability detection tasks while extending their approach to address the specific challenges presented by each project's unique security context.

The scholarly discourse around LLMs in security contexts has evolved rapidly, with Almeida et al. (2024) demonstrating practical implementations in IDE integration and Wu et al. (2023) exploring their effectiveness in vulnerability remediation. However, as Yang et al. (2023) note in their comprehensive review of 146 LLM studies, significant questions remain about their application in security-critical contexts. This research addresses a gap in current understanding: the practical integration of LLM capabilities within existing code review workflows for security vulnerability detection across diverse programming languages and security domains.

The findings of this research have significant implications for both theoretical understanding and practical application of AI-assisted security analysis. For development teams, this study provides empirically grounded insights into integrating specific LLM-based tools into existing code review processes, with particular attention to the varying requirements of web frameworks, logging infrastructure, and system utility security reviews.

By evaluating the capability of current-generation LLMs to identify security vulnerabilities during code review across these three distinct open-source projects, this research aims to connect theoretical concepts with practical execution. The results may inform both tool selection and process improvements in software security practices, contributing to the broader scholarly discussion of AI-assisted software security while providing actionable insights for organizations seeking to enhance their security review capabilities.

2. Research Method

The methodological framework for this study employs a systematic, multi-phase approach to evaluating Large Language Models' capabilities in security vulnerability detection. This research implements a controlled testing environment across three distinct technological domains by drawing on the analytical frameworks established by Yu et al. (2024) and extending them to address specific security review contexts.

2.1 Methodology Overview

2.1.1 Selection Process for Vulnerable Code Samples

The research centers on three historically significant Common Vulnerabilities and Exposures (CVEs) that represent diverse security challenges across different programming languages and paradigms:

- **CVE-2022-28346**; CVSS 9.8; Django 4.0.3: SQL Injection vulnerability in *django/db/models/sql/query.py*
  https://nvd.nist.gov/vuln/detail/cve-2022-28346
- **CVE-2021-44228**; CVSS 10.0; Log4j 2.14.1: Remote Code Execution vulnerability in *core/src/main/java/org/apache/logging/log4j/core/lookup/JndiLookup.java*
  https://nvd.nist.gov/vuln/detail/cve-2021-44228
- **CVE-2021-3156;** CVSS 7.8; Sudo 1.9.5p1: Buffer Overflow vulnerability in *plugins/sudoers/sudoers.c*
  https://nvd.nist.gov/vuln/detail/cve-2021-3156

These vulnerabilities were selected based on their documented impact and severity, the availability of pre-patch source code via GitHub, their representation of unique vulnerability classes across multiple programming languages, and the existence of the vulnerability in a single source code file to simplify the experimentation process. This selection allows for the evaluation of LLM performance across varied technical contexts while maintaining controlled testing conditions.

These vulnerabilities represent different classes of security flaws that continue to plague modern software development, and together, they provide a comprehensive test bed for evaluating LLM capabilities across different security domains and programming paradigms. The Log4Shell vulnerability (CVE-2021-44228) is one of the most severe and widespread security incidents in recent history, earning a CVSS score of 10.0. The vulnerability's exploitation pattern—unrestricted JNDI lookups enabling remote code execution—demonstrates how seemingly benign logging functionality can be weaponized for malicious purposes. This vulnerability highlights the risks of using third-party dependencies and the critical importance of validating external inputs.

The Sudo vulnerability (CVE-2021-3156) represents a classic buffer overflow vulnerability in C, a programming language where memory management is a constant security challenge. With a CVSS score of 7.8, this vulnerability allowed local privilege escalation by manipulating command-line arguments, demonstrating how subtle implementation details in security-critical software can lead to significant breaches. The vulnerability's presence in Sudo, a fundamental Unix/Linux security utility, underscores the importance of rigorous security review for privileged system components and the ongoing relevance of memory safety concerns in systems programming.

The Django SQL injection vulnerability (CVE-2022-28346), scoring 9.8 on CVSS, illustrates the persistent challenge of secure data handling in modern web frameworks. Despite Django's robust security architecture and built-in protections against SQL injection, this vulnerability emerged from complex interactions between the ORM's query generation system and certain types of field lookups. This case demonstrates how even frameworks designed with security in mind can harbor subtle vulnerabilities, particularly at the intersection of convenience features and security boundaries.

These vulnerabilities were selected for their clear documentation and reproducibility, making them ideal candidates for controlled experimentation. Each represents a distinct security lesson: Log4Shell emphasizes the importance of strict input validation and secure defaults, the Sudo vulnerability highlights the critical nature of memory safety in privileged operations, and the Django case demonstrates the complexity of dealing with SQL query sanitization in modern web application frameworks.

2.1.2 LLM Configuration and Selection

The study employs three current-generation Large Language Models, each representing different approaches to code analysis:

- GitHub Copilot using GPT-4o: Selected for its specialized training in code analysis and integration with development environments
- Google Gemini Advanced 1.5 Pro: Chosen for its advanced reasoning capabilities and broad knowledge base
- Claude 3.5 Sonnet: Selected for its demonstrated proficiency in code understanding and security analysis

Each model's configuration remains consistent throughout testing to ensure reproducibility and valid comparative analysis.

2.1.3 Testing Protocol Development

The testing protocol implements a three-phased approach to vulnerability detection using the prompts below:

a.) Initial Code Review: Each LLM analyzes the vulnerable code without specific security prompting: "*Review this code for overall quality and any potential issues.*"

b.) Guided Security Analysis: Targeted prompts direct the LLMs to identify potential security issues: "*Review this code for any potential security vulnerabilities.*"

c.) Vulnerability-Specific Analysis: Focused evaluation of each model's ability to identify the specific vulnerability class present in each case: "*Review this code for any potential buffer overflows.*"

2.2 Testing Environment Setup

2.2.1 Code Repository Preparation

Vulnerable code samples are retrieved from GitHub and isolated from their respective repositories to create controlled testing conditions. Each sample is preserved in its pre-patch state, maintaining the context necessary for vulnerability detection while eliminating potential confounding variables from surrounding codebase changes.

2.2.2 LLM Configuration and Prompt Engineering

Drawing on Jensen et al.'s (2024) findings regarding prompt effectiveness in security contexts, the research employs a graduated prompting strategy if vulnerabilities are not identified by the initial prompt(s):

- General Code Review: Open-ended analysis of code quality and potential issues
- Security-Focused Review: Specific prompts for security vulnerability detection
- Targeted Vulnerability Analysis: Focused examination of specific vulnerability classes

2.2.3 Vulnerability Validation Framework

The validation process implements a three-tiered assessment structure:

- Detection Accuracy: Ability to identify the presence of a vulnerability
- Classification Precision: Accuracy in categorizing the type of vulnerability
- Context Understanding: Comprehension of the vulnerability's potential impact and exploitation vectors

2.3 Data Collection Approach

2.3.1 Vulnerability Detection Metrics

The research tracks multiple quantitative and qualitative metrics:

- True Positive Rate: Correct vulnerability identifications
- Detection Precision: Accuracy of vulnerability classification
- Analysis Depth: Comprehensiveness of security insights

2.3.2 Performance Measurements

Performance evaluation encompasses:

- Detection Outcome: Whether the LLM correctly identified the known vulnerability
- Detection Specificity: Level of detail in vulnerability description and understanding of the security impact

- Prompt Efficiency: Number of interactions required before successful detection

2.3.3 Error Rate Tracking

Error analysis focuses on:

- Misclassification Patterns: Systematic errors in vulnerability categorization
- Context Failures: Instances where environmental context was misunderstood
- False Negatives: Missed vulnerabilities and their characteristics

This methodological framework enables systematic evaluation of LLM capabilities in security vulnerability detection while maintaining scholarly rigor and reproducibility. The approach balances practical testing requirements with theoretical foundations established in current literature, providing a structured basis for analyzing the feasibility of LLM integration into security-focused code review processes.

3. Findings and Discussion

3.1 LLM Performance Analysis

3.1.1 Overall Detection Capabilities

The experimental results reveal intricate patterns in how Large Language Models approach security vulnerability detection across different programming languages and vulnerability types. The investigation, centered on three historically significant vulnerabilities—Log4Shell remote code execution (CVE-2021-44228), Sudo buffer overflow (CVE-2021-3156), and Django SQL injection (CVE-2022-28346)—provides compelling insights into the capabilities and limitations of current-generation LLMs in security analysis contexts.

All three models—GitHub Copilot, Google Gemini, and Claude—demonstrated sophisticated capabilities in identifying critical security vulnerabilities, though with notable variations in their analytical approaches and detection methodologies. A particularly significant finding emerged in the universal success rate for Log4Shell vulnerability detection, with all three models identifying the vulnerability in their initial analysis pass. This consistency suggests robust pattern recognition capabilities for well-

documented, high-impact vulnerabilities, likely attributed to the extensive coverage and discussion of Log4Shell in security literature and documentation.

The models' detection capabilities showed interesting variations across different programming languages. For Java-based vulnerabilities (Log4Shell), all models demonstrated strong initial detection rates, possibly reflecting the structured nature of Java code and the extensive documentation of Java-based security vulnerabilities in training data. Python vulnerabilities (Django) showed more variable detection patterns, while C-based vulnerabilities (Sudo) required more specific prompting for successful identification, suggesting potential gaps in lower-level security analysis capabilities.

3.1.2 Comparative Analysis Between Different LLMs

The research revealed distinct analytical patterns and capabilities across the three models, each demonstrating unique strengths and limitations in their approach to vulnerability detection:

GitHub Copilot:

- Demonstrated exceptional performance in identifying the Log4Shell vulnerability, providing detailed technical analysis including:
    - Specific identification of unrestricted JNDI lookups
    - Recognition of potential remote code execution vectors
    - Detailed remediation strategies, including protocol restrictions
- Required more targeted prompting for buffer overflow detection
- Showed strong code quality analysis capabilities but sometimes at the expense of security-specific insights
- Provided practical, implementation-focused remediation suggestions

Google Gemini:

- Exhibited comprehensive contextual analysis capabilities, particularly evident in:
    - Broader security implication assessment
    - Detailed architectural impact analysis
    - Integration of security best practices in recommendations
- Successfully identified Log4Shell vulnerability with initial prompt

- Provided extensive security context but occasionally at the cost of specificity
- Demonstrated strong performance in identifying architectural security patterns

Claude:

- Showed notable efficiency in initial vulnerability detection:
    - Successful identification of both Log4Shell and Sudo vulnerabilities in first-pass analysis
    - Precise technical detail in vulnerability descriptions
    - Conservative but accurate assessment methodology
- Provided balanced analysis between security and functionality
- Demonstrated a strong correlation between detection confidence and accuracy
- Excelled in providing context-aware security recommendations

3.1.3 Vulnerability Type Effectiveness

The research revealed nuanced patterns in detection effectiveness across different vulnerability classes, with each model demonstrating distinct capabilities in identifying and analyzing specific types of security concerns:

Remote Code Execution (Log4Shell): The Log4Shell vulnerability served as a compelling case study in LLM detection capabilities, revealing sophisticated pattern recognition across all three models. GitHub Copilot's analysis was particularly noteworthy, providing a detailed technical breakdown:

> "*The code performs JNDI lookups without any restrictions on the lookup string [...] This enables attackers to execute remote code through malicious JNDI lookups [...] There are no input validation checks or protocol restrictions.*"

This level of technical precision suggests strong capability in identifying architectural security patterns, particularly in Java-based systems. Google Gemini's analysis provided additional contextual depth:

> "*The primary concern with JNDI lookups is the potential for remote code execution (RCE) vulnerabilities. If an attacker can control the JNDI URL being looked up, they could potentially execute arbitrary code on the server.*"

The consistency in detection across all models suggests that well-documented, high-impact vulnerabilities create strong pattern recognition signatures that LLMs can readily identify and analyze.

Buffer Overflow (Sudo): The analysis of buffer overflow detection revealed more complex patterns, with varying degrees of success across models and prompting strategies:

- Initial Detection:
    - Claude identified potential buffer overflow risks in the first analysis pass
    - GitHub Copilot required specific prompting for identification
    - Google Gemini provided general security concerns before identifying the specific vulnerability

Claude's analysis after the first prompt demonstrated acuity in this area:

> "*Potential buffer overflow in size calculation if there are many arguments [...] Pointer arithmetic that could be unsafe if string literal size changes.*"

This granular understanding of memory safety issues suggests strong capabilities in analyzing lower-level security concerns. However, the need for specific prompting with other models indicates potential limitations in baseline detection capabilities for memory-related vulnerabilities.

SQL Injection (Django): The Django SQL injection vulnerability analysis revealed interesting patterns in how models approach web application security:

- Initial Analysis:
    - All models initially focused on code quality and structure
    - Security implications emerged more clearly with targeted prompting
    - Specific vulnerability identification varied by model

Google Gemini's analysis evolved significantly with security-focused prompting:

> "*The Query class builds SQL queries based on Django QuerySet operations. The code uses parameterization extensively, which is the recommended way to prevent SQL injection.*"

3.1.4 False Positive/Negative Evaluation

The experimental results revealed sophisticated patterns in error rates across different models and vulnerability types:

False Positives Analysis

General Code Review Context:

- Higher rates of potential security issue identification
- Often focused on best practices rather than actual vulnerabilities
- Varied by programming language and framework

The models demonstrated different tendencies in false positive generation:

GitHub Copilot:

- Showed higher sensitivity to potential security issues
- Often flagged code quality issues as security concerns
- Provided detailed but sometimes overly cautious analysis

Google Gemini:

- Demonstrated balanced detection with moderate false positive rates
- Showed a strong contextual understanding of vulnerability assessment
- Provided comprehensive security context for findings

Claude:

- Exhibited conservative detection patterns
- Showed lower false positive rates in the initial analysis
- Maintained high precision in vulnerability identification

False Negatives Analysis

The pattern of false negatives revealed important insights into model limitations:

Buffer Overflow Detection:

- Higher false negative rates in the initial analysis
- Improved significantly with security-focused prompting

- Varied by code complexity and context

SQL Injection:

- Initial false negatives in framework-specific contexts
- Improved detection with explicit security focus
- Strong pattern recognition, once properly prompted

The research revealed that false negative rates were significantly influenced by:

- Programming language complexity
- Framework-specific implementations
- Security context availability
- Prompt engineering effectiveness

3.2 Implementation Insights

3.2.1 Prompt Engineering Effectiveness

The research demonstrated the critical importance of prompt engineering in vulnerability detection, revealing complex relationships between prompt structure and detection accuracy. The following three prompts were executed in order, with the latter prompts only being used if the preceding prompt(s) were unable to identify the target vulnerability:

General Code Review Prompts (Prompt 1):

"*Review this code for overall quality and any potential issues.*"

- Generated broader security considerations
- Often missed specific vulnerabilities
- Provided valuable contextual analysis

Security-Focused Prompts (Prompt 2):

"*Review this code for any potential security vulnerabilities.*"

- Improved specific vulnerability detection
- Enhanced technical precision in analysis
- Reduced false positive rates

Vulnerability-Specific Prompts (Prompt 3):

"*Review this code for any potential buffer overflows.*"

- Highest detection accuracy for targeted vulnerabilities
- Reduced false negative rates
- Potentially missed other security issues

The effectiveness of different prompting strategies varied by model:

GitHub Copilot:

- Responded well to specific technical prompts
- Showed strong improvement with security-focused prompting
- Maintained consistent analysis quality across prompt types

Google Gemini:

- Provided comprehensive analysis regardless of the prompt type
- Showed strong contextual understanding across prompts
- Benefited from security-specific prompting for detailed analysis

Claude:

- Demonstrated strong baseline security analysis
- Showed consistent performance across prompt types
- Provided detailed technical analysis with minimal prompting

3.2.2 Integration Challenges

The research revealed several interconnected challenges in integrating LLM-based vulnerability detection into existing code review workflows, with language-specific considerations emerging as a primary concern. The models demonstrated varying levels of detection accuracy across different programming languages, necessitating careful calibration of analysis approaches based on the target codebase's linguistic context. This variability manifests particularly in framework-specific security pattern recognition, where the models' ability to identify vulnerabilities often depends on their familiarity with specific framework architectures and common security patterns within those contexts. The research further identified distinct prompting requirements across different

programming languages, suggesting that effective implementation requires language-specific prompt engineering strategies to optimize detection capabilities.

Workflow integration presents another dimension of complexity centered on achieving an optimal balance between automated LLM analysis and manual security review processes. Integrating LLM-based detection systems with existing security tools requires careful consideration of workflow dynamics and tool interoperability. At the same time, the standardization of prompting strategies across different review contexts emerges as a critical factor in maintaining consistent security analysis quality. The challenge of standardization is particularly relevant in organizations dealing with numerous codebases and multiple programming languages. Here, maintaining consistent security analysis quality across different technical contexts is critical.

Technical implementation considerations further complicate the integration landscape, encompassing challenges in API integration, response processing, and performance optimization. The research indicates that successful implementation requires sophisticated API integration strategies to handle varying response patterns across LLM platforms while maintaining consistent security analysis quality. The processing and analysis of model responses present additional challenges, particularly in contexts requiring rapid security assessment and remediation guidance. Performance optimization emerges as a critical consideration, particularly in large-scale code review workflows where analysis speed and resource utilization efficiency become key factors in successful implementation.

3.2.3 Resource Requirements

The analysis revealed a complex landscape of resource requirements necessary for effective LLM implementation in security review processes. Computational resource demands were surprisingly modest, with the research demonstrating minimal latency impact on existing review workflows across all tested models. The consistent response times observed across GitHub Copilot, Google Gemini, and Claude suggest robust scalability potential, particularly in enterprise-level implementation contexts. This computational efficiency indicates that organizations can integrate these tools without

significant infrastructure overhaul, though careful attention to system architecture remains crucial for optimal performance.

Human resource considerations proved more nuanced, requiring a sophisticated blend of technical expertise and security knowledge. The research revealed that effective implementation demands specialized knowledge in prompt engineering, particularly for optimizing vulnerability detection across different programming languages and security contexts. This expertise requirement extends beyond traditional security knowledge, encompassing an understanding of LLM behavior patterns and response characteristics. Additionally, organizations must invest in comprehensive training programs to ensure review teams can effectively validate and interpret model outputs, suggesting a need for ongoing professional development in both security analysis and LLM interaction methodologies.

Infrastructure requirements presented a third critical dimension centered on integrating API access systems and response processing frameworks. The research indicates that successful implementation necessitates robust API management systems capable of handling multiple model interactions while maintaining security and performance standards. These systems must be complemented by sophisticated response processing frameworks that can effectively parse and categorize security findings, while security result management systems are essential for tracking and validating model outputs across different review contexts.

3.2.4 Cost Considerations

The research uncovered a multifaceted cost structure associated with LLM implementation in security review processes, encompassing direct and indirect financial impacts. Direct costs manifest primarily through API usage fees, which vary significantly across LLM platforms and usage patterns. These fundamental expenses are augmented by substantial investment requirements in integration development, including initial implementation costs and ongoing maintenance needs. The research also highlighted the significant expenditure necessary for comprehensive training and documentation systems, essential for ensuring effective tool utilization across security review teams.

Indirect costs emerged as equally significant, though more challenging to quantify precisely. The investigation of false positives represents a particularly notable indirect cost, requiring dedicated security analyst time to validate and verify model outputs. This challenge is compounded by the ongoing need for prompt optimization efforts, which demand continuous refinement based on detection accuracy and evolving security concerns. Security validation overhead further contributes to these indirect costs, necessitating a careful balance between automated detection and human verification processes.

Despite these cost considerations, the research revealed substantial efficiency gains that may offset initial and ongoing expenses. Implementing LLM-based security review processes demonstrated a significant reduction in manual review time, particularly for well-documented vulnerability patterns. Early vulnerability detection capabilities suggest potential cost savings through reduced security incident response needs, while improved remediation guidance may lower security maintenance costs. These efficiency improvements, coupled with enhanced detection capabilities, suggest that thoughtfully implemented LLM-based security review systems may provide a compelling return on investment despite substantial initial and ongoing costs.

3.3 Security Impact Assessment

The experimental findings demonstrate compelling evidence for the transformative potential of LLM integration in security vulnerability detection processes, with sophisticated patterns emerging across different analytical contexts and vulnerability classifications. The research reveals particularly noteworthy success in detecting well-documented vulnerabilities, as evidenced by the uniform identification of Log4Shell (CVE-2021-44228) across all tested models. This consistent performance suggests robust pattern recognition capabilities for high-impact security issues, while the variable detection rates observed for buffer overflow and SQL injection vulnerabilities illuminate important nuances in the models' analytical capabilities. The progression from initial detection rates of 33% to 100% with targeted prompting for the Sudo buffer overflow (CVE-2021-3156) and Django SQL injection (CVE-2022-28346) vulnerabilities indicates significant potential for enhanced detection through refined implementation strategies.

The temporal efficiency gains revealed through the research suggest substantial potential for optimizing security review processes through strategic LLM integration. The models demonstrated remarkable capability in providing comprehensive security analyses within timeframes that would challenge human reviewers, with particularly strong performance in rapid vulnerability triage and detailed technical context generation. This efficiency manifests most notably in the immediate identification of well-documented vulnerabilities and the simultaneous generation of detailed remediation strategies, suggesting significant potential for reducing initial security screening time while maintaining analytical depth. The research further indicates that LLM integration allows for more strategic allocation of security expertise, particularly in areas requiring nuanced understanding or complex decision-making.

The experimental results reveal sophisticated patterns in risk reduction potential through LLM integration, with particularly strong performance in identifying framework-specific vulnerabilities and language-specific security concerns. The models demonstrated variable but generally robust capabilities across different programming contexts, with notable strength in high-level language analysis and improved detection rates through language-specific prompt optimization. This capability highlights the potential to enhance security by integrating LLMs into code review processes, particularly when combining automated analysis with human expertise.

The research conclusively demonstrates that while LLMs offer considerable promise in augmenting security-focused code reviews, their effectiveness varies significantly based on vulnerability type, programming language, and implementation approach. This variability underscores the importance of developing sophisticated integration strategies that carefully balance automated analysis capabilities with human security expertise. Success in implementation requires meticulous attention to prompt engineering strategies, language-specific optimization, integration workflow design, and security validation processes. The findings suggest that when properly implemented, LLM integration can substantially enhance security review processes, though ongoing optimization of detection strategies remains crucial for maintaining effectiveness across evolving security landscapes.

4. Recommendations and Implications for Future Research

4.1 Recommendations for Practice

The experimental findings suggest several concrete recommendations for organizations seeking to integrate LLMs into their security review processes. The demonstrated success of GitHub Copilot, Google Gemini, and Claude in identifying critical vulnerabilities indicates that organizations should adopt a multi-model approach to security analysis, leveraging the complementary strengths of different LLMs to enhance detection capabilities. This strategic integration should incorporate carefully crafted prompting hierarchies that progress from general code review to targeted security analysis, particularly when examining code for potential buffer overflows, SQL injections, and other subtle security vulnerabilities that might not be obvious to a human reviewer.

Organizations should establish robust validation frameworks that combine LLM analysis with traditional security tools and human expertise. The research demonstrates that while LLMs can identify well-documented vulnerabilities, their effectiveness is optimized when integrated into comprehensive security review frameworks that include static analysis tools, dynamic testing, and expert human oversight. This layered approach ensures that the pattern recognition capabilities of LLMs complement, rather than replace, existing security analysis methodologies.

4.2 Implications for Future Research

The findings of this study illuminate several critical directions for future research in applying LLMs to security-focused code review. A primary avenue for investigation lies in the potential for fine-tuning existing open-source models with security-specific domain knowledge. While current models demonstrate strong capabilities in identifying known vulnerabilities, their performance variability across different vulnerability types suggests that targeted fine-tuning with comprehensive security datasets could enhance detection precision. Future studies should explore the development of specialized security-focused variants of existing models, potentially incorporating knowledge from vulnerability

databases, security advisories, and patch repositories to create more robust analytical tools.

The research also reveals a compelling need to evaluate model performance against lesser-known security vulnerabilities, particularly those that may not be well-represented in current training datasets. While the studied models excelled at identifying high-profile vulnerabilities like Log4Shell, their capabilities in detecting novel or obscure security issues remain largely unexplored. Future research should systematically examine model performance against broader vulnerabilities, including those specific to emerging technologies and specialized frameworks. This investigation would advance analysts' understanding of LLM generalization capabilities in security contexts while potentially revealing new approaches to enhancing their vulnerability detection abilities through architectural innovations and training methodologies.

A significant limitation of the current research lies in its focus on isolated source code files, suggesting a critical need for future studies to examine LLM performance in more complex, interconnected codebases. Modern software systems typically comprise intricate networks of dependencies, microservices, and distributed components, where security vulnerabilities may manifest through subtle interactions between multiple code modules. Future research should investigate how LLMs perform when analyzing entire software systems, including their ability to trace vulnerability patterns across module boundaries, identify security implications in architectural dependencies, and understand context-dependent security risks that emerge from component interactions. This expanded scope would provide crucial insights into the scalability and practical applicability of LLM-based security analysis in enterprise-scale software development environments.

5. Conclusion

The challenge of spotting critical security vulnerabilities during routine code reviews presents a major obstacle in software development, especially as the complexity and scope of potential vulnerabilities continue to expand. This research began with a basic question about the feasibility of leveraging Large Language Models to enhance security vulnerability detection during code reviews, specifically examining whether these models could effectively identify security issues that human reviewers had previously missed. Examining three distinct vulnerability types across multiple programming languages, this study has illustrated both the remarkable capabilities and significant limitations of current-generation LLMs for use in code reviews.

The experimental findings support the thesis that LLMs can effectively augment human code reviews to identify security vulnerabilities, though with important caveats. The consistent success in identifying well-documented vulnerabilities like Log4Shell across all tested models demonstrates robust pattern recognition capabilities, while the more variable performance in detecting buffer overflows and SQL injection vulnerabilities reveals the importance of prompt engineering strategies and careful implementation approaches. The research highlighted the models' ability to provide a detailed security analysis and remediation strategy far quicker than a human could, suggesting significant potential for enhancing the efficiency and comprehensiveness of the code review processes.

The ultimate assessment of LLM viability for security-focused code reviews is cautiously optimistic and points toward a future where artificial intelligence augments and enhances human security expertise, potentially leading to more robust and comprehensive software security practices. The demonstrated capabilities in quickly identifying potential vulnerabilities and providing detailed remediation guidance indicate that LLMs can greatly enhance the code review process. However, the variation in detection capabilities across different vulnerability classes and programming languages underscores the importance of developing integrated approaches that combine LLM analysis with traditional security tools and human expertise.

Appendix

A. Experimental Response Data

A.1 CVE-2021-44228 (Log4j 2.14.1)

This section presents the detailed responses from each model when analyzing the Log4Shell vulnerability in *JndiLookup.java*.

A.1.1 Initial Detection Success Rates

Model	Detection with First Prompt	Accuracy of Analysis
GitHub Copilot	Yes	High
Google Gemini	Yes	High
Claude	Yes	High

A.1.2 Key Detection Elements

All three models successfully identified:

- Unrestricted JNDI lookups as the core vulnerability
- Potential for remote code execution
- Lack of input validation
- Missing protocol restrictions

A.1.3 Notable Response Variations

GitHub Copilot provided the most technically detailed analysis, specifically highlighting:

"*The code performs JNDI lookups without any restrictions on the lookup string [...]*
This enables attackers to execute remote code through malicious JNDI lookups [...]
There are no input validation checks or protocol restrictions."

A.2 CVE-2021-3156 (Sudo 1.9.5p1)

This section details the models' analysis of the buffer overflow vulnerability in *sudoers.c*.

A.2.1 Detection Success by Prompt Level

Model	Prompt 1	Prompt 2	Prompt 3
GitHub Copilot	No	No	Yes
Google Gemini	No	No	Yes
Claude	Yes	-	-

A.2.2 Detection Patterns

1. Initial prompts primarily yielded general code quality observations
2. Security-focused prompts improved detection but still missed the specific vulnerability
3. Buffer overflow-specific prompts led to successful identification in most cases

A.2.3 Key Variations in Analysis

Claude's initial response identified:

"Potential buffer overflow in size calculation if there are many arguments
Pointer arithmetic that could be unsafe if string literal size changes"

A.3 CVE-2022-28346 (Django 4.0.3)

This section examines the models' performance in identifying the SQL injection vulnerability in *query.py*.

A.3.1 Detection Success Rates

Model	First Prompt Detection	Second Prompt Detection
GitHub Copilot	No	Yes
Google Gemini	No	Yes
Claude	Yes	Yes

A.3.2 Analysis Components

Common elements identified across models:

- Need for parameterized queries
- Risks in raw SQL execution
- Input validation requirements
- Query sanitization recommendations

B. Comparative Analysis

B.1 Detection Efficiency

Detection speed and accuracy varied significantly:

1. Log4Shell: Immediate detection by all models
2. Sudo Buffer Overflow: Variable detection requiring specific prompting
3. Django SQL Injection: Mixed initial detection with improvement on security-focused prompts

B.2 Analysis Depth

Qualitative assessment of analysis depth:

Aspect	GitHub Copilot	Google Gemini	Claude
Technical Detail	High	Medium	High
Context Understanding	Medium	High	High
Remediation Guidance	High	Medium	High
False Positive Rate	Low	Low	Low

B.3 Response Pattern Analysis

1. **Initial Responses**
   - Focus on code quality and structure
   - Variable security awareness
   - Comprehensive documentation review

2. **Security-Focused Responses**
    - Increased vulnerability detection
    - More specific technical details
    - Enhanced remediation suggestions
3. **Vulnerability-Specific Responses**
    - Highest accuracy rates
    - Most detailed technical analysis
    - Concrete mitigation strategies

C. Methodological Notes

C.1 Prompt Strategy

The three-tiered prompting strategy employed:

1. General code review
2. Security-focused review
3. Vulnerability-specific review

C.2 Response Evaluation Criteria

Responses were evaluated based on:

- Accuracy of vulnerability identification
- Depth of technical analysis
- Quality of remediation suggestions
- False positive/negative rates
- Comprehensiveness of security context

C.3 Limitations

Notable limitations in the experimental approach:

1. Limited sample size of vulnerabilities
2. Potential prompt sensitivity
3. Model version dependencies
4. Context window limitations
5. Temporal nature of model knowledge

References

Almeida, Y., Albuquerque, D., Dantas Filho, E., Muniz, F., de Farias Santos, K., Perkusich, M., Almeida, H., & Perkusich, A. (2024). AICodeReview: Advancing code quality with AI-enhanced reviews. *SoftwareX, 26*, Article 101677. https://doi.org/10.1016/j.softx.2024.101677

Alrashedy, K., & Aljasser, A. (2023). Can LLMs patch security issues? *arXiv preprint*. https://doi.org/10.48550/arXiv.2312.00024

Jensen, R. I. T., Tawosi, V., & Alamir, S. (2024). Software vulnerability and functionality assessment using LLMs. *arXiv preprint*. https://doi.org/10.48550/arXiv.2403.08429

Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., Babkin, P., & Shah, S. (2023). How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 1282-1294). Association for Computing Machinery. https://doi.org/10.1145/3597926.3598135

Yang, Z., Sun, Z., Yue, T. Z., Devanbu, P., & Lo, D. (2023). Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *Journal of the ACM, 37*(4), Article 1. https://doi.org/10.48550/arXiv.2403.07506

Yin, X., & Ni, C. (2024). Multitask-based evaluation of open-source LLM on software vulnerability. *arXiv preprint*. https://doi.org/10.48550/arXiv.2404.02056

Yu, J., Liang, P., Fu, Y., Tahir, A., Shahin, M., Wang, C., & Cai, Y. (2024). Security code review by LLMs: A deep dive into responses. *arXiv preprint*. https://doi.org/10.48550/arXiv.2401.16310

Zhou, X., Cao, S., Sun, X., & Lo, D. (2024). Large language model for vulnerability detection and repair: Literature review and the road ahead. *arXiv preprint*. https://doi.org/10.48550/arXiv.2404.02525