Security Analysis of AI-Generated Source Code: A Comparative Study of Vulnerability Patterns in Large Language Model (LLM) Generated and Human-Written Programs

Author's Name: **Kedar Ganesh Rane** Department of Cyber Security
Chikitsak Samuha's Patkar -Varde College, Mumbai 400 062, India
Email ID: kedar.rane.2378@gmail.com

Abstract — AI coding assistants based on Large Language Models (LLMs) are now widely used to generate source code, but the security quality of that code is not guaranteed. This paper proposes a design-and-evaluation framework for comparing the security of LLM-generated programs with human-written programs. Programs for identical task specifications are produced by popular AI coding assistants and by human developers, then analysed with static application security testing (SAST) tools and manual verification. The study focuses on five weakness categories: SQL Injection, Cross-Site Scripting (XSS), Broken Authentication, Insecure File Handling, and Hardcoded Credentials, mapped to CWE and OWASP classifications. Findings are compared using vulnerability density, vulnerable-program rate, severity distribution, and scanner precision. Static analysis is treated as an aid rather than a final judge: flagged issues are manually verified before being counted. The paper also discusses ethical use of AI tools, licensing, threats to validity, and recommendations for secure AI-assisted development.

Index Terms — AI-Generated Code, Large Language Models, Secure Coding, Static Code Analysis, Vulnerability Patterns, SQL Injection, XSS, CWE, OWASP, Software Security.

I. INTRODUCTION

AI coding assistants such as GitHub Copilot, ChatGPT, and Claude can produce working code from natural-language prompts within seconds. This speeds up development, but the models are trained on large collections of existing code, which may itself contain insecure patterns. A model optimises for code that looks plausible and functional; it does not inherently guarantee that the code is secure.

Earlier studies have examined this concern. Pearce et al. evaluated code suggested by GitHub Copilot across security-relevant scenarios and found that a significant share of the suggestions contained vulnerabilities [1]. Perry et al. reported in a user study that participants with access to an AI assistant wrote less secure code in several tasks and were more confident in its security than they should have been [2]. These findings motivate a structured comparison against human-written baselines across multiple assistants and multiple vulnerability classes.

This paper proposes such a comparison. Its emphasis is on a reproducible methodology: controlled prompts, comparable human baselines, standard vulnerability taxonomies (CWE, OWASP), automated scanning, and manual verification. AI code is treated as untrusted input that must be reviewed like any third-party contribution.

II. PROBLEM STATEMENT

Security evaluation of AI-generated code faces four core problems: (a) adoption speed — developers use AI-generated code faster than organisations can build review practices around it; (b) over-trust — generated code appears clean and confident, which can discourage careful review; (c) measurement difficulty — static tools produce false positives and false negatives, and results differ by tool, language, and prompt wording; and (d) non-determinism — the same prompt can yield different code across runs and model versions, making results hard to reproduce. An effective study must therefore control prompts, repeat generation, combine more than one analysis tool, and verify findings manually.

III. AIM AND OBJECTIVES

The aim is to compare the security quality of LLM-generated and human-written programs and to identify recurring vulnerability patterns. The objectives are: (1) define a set of realistic programming tasks likely to involve security-sensitive functionality; (2) generate solutions using popular AI coding assistants; (3) obtain comparable human-written solutions; (4) analyse all programs using static analysis and secure-coding checklists; (5) compare vulnerability

frequency, type, and severity; and (6) recommend best practices for the secure use of AI-assisted development.

IV. LITERATURE REVIEW

A. Secure Software Development

Secure development guidance such as the NIST Secure Software Development Framework (SP 800-218) recommends practices including secure design, code review, and vulnerability testing throughout the lifecycle [3]. OWASP publishes the Top 10 list of critical web application risks, including injection, broken authentication, and cryptographic failures [4]. MITRE's Common Weakness Enumeration (CWE) provides a standard vocabulary for classifying software weaknesses [5].

B. LLMs for Code Generation

Code-focused language models are trained on large code corpora and can complete or generate functions from natural-language descriptions [6]. Their output quality is usually evaluated for functional correctness, whereas security is a separate property that is not guaranteed by passing tests.

C. Security of AI-Generated Code

Prior work has assessed AI-generated code with scenario-based prompts and static analysis [1], and has studied how developers behave when using AI assistants [2]. Results indicate that vulnerabilities do occur in generated code, but their rate depends on the task, language, model, and prompt. Conclusions from one tool or model version do not automatically transfer to others.

D. Research Gap

Many studies focus on a single assistant or a narrow set of scenarios. A gap remains in comparing several assistants against a human-written baseline using the same task specifications, the same analysis tools, and manual verification of findings, while reporting the error rates of the scanners themselves. This paper addresses that gap with a controlled, repeatable design.

V. RESEARCH QUESTIONS AND HYPOTHESES

RQ1: Which vulnerability types occur most often in LLM-generated code?

RQ2: How does the vulnerability rate of LLM-generated code compare with human-written code for the same tasks?

RQ3: Does prompt wording (plain versus security-aware prompts) change the vulnerability rate?

RQ4: How reliable are static analysis tools for detecting these weaknesses, and how many findings survive manual verification?

H0: There is no difference in vulnerability density between LLM-generated and human-written programs.

H1: LLM-generated programs show a different (expected: higher) vulnerability density than human-written programs for security-sensitive tasks.

VI. VULNERABILITY CATEGORIES UNDER STUDY

Category	Related CWE	Typical Insecure Pattern	Detection Approach
SQL Injection	CWE-89	String-concatenated queries	SAST + manual review
Cross-Site Scripting	CWE-79	Unescaped user input in HTML output	SAST + manual review
Broken Authentication	CWE-287, 256, 307	Weak or plain-text password handling, no lockout, weak sessions	Checklist + manual review
Insecure Fil	CWE-22, 434	Path traversal, unrestricted file upload	SAST + manual review

e Handling			
Hardcoded Credentials	CWE-798	Keys or passwords embedded in source	SAST + secret scanning

Table I. Weakness categories and detection approach

Availability of automated detection varies by category; authentication logic in particular often needs manual assessment.

VII. RESEARCH METHODOLOGY

The study follows a design-and-evaluation approach. It uses only code written for the study; no third-party private code is analysed.

A. Task and Prompt Design

A set of 10–15 small web/back-end tasks is defined, for example: user login, user registration, search by keyword, file upload and download, comment posting, and configuration loading that requires an API key. Each task has a written specification and is implemented in one or two languages (for example Python and JavaScript). Each task is prompted in two ways: a plain prompt describing functionality only, and a security-aware prompt that additionally asks for secure code.

B. AI Code Generation

Each task is submitted to the selected AI coding assistants using identical prompts. Each prompt is run several times (for example, five) to account for non-determinism. Assistant name, model version, date, prompt, and raw output are recorded. Generated code is saved unedited.

C. Human-Written Baseline

Human solutions are written by volunteer developers (for example, students or junior developers) from the same specifications, without AI assistance. Participation is voluntary and anonymised. Developers' experience level is recorded because it affects the comparison.

D. Static and Manual Analysis

All programs are scanned with more than one SAST tool, for example Bandit (Python), Semgrep, and CodeQL, and with a secret scanner for hardcoded credentials. Every reported finding is then manually reviewed and labelled as a true positive or a false positive. A secure-coding checklist derived from OWASP is used for weaknesses that scanners handle poorly, such as authentication logic. Two reviewers should label independently, with disagreements resolved by discussion.

E. Evaluation Metrics

- **Vulnerable-program rate** = programs with at least one verified vulnerability / total programs
- **Vulnerability density** = verified vulnerabilities per 1,000 lines of code (KLOC)
- **Severity distribution** using CWE/CVSS-style ratings
- **Scanner precision** = TP / (TP + FP)
- **Scanner recall** (estimated on a manually audited subset) = TP / (TP + FN)
- **Inter-reviewer agreement** (for example, Cohen's kappa)

Group differences are tested with appropriate statistical tests (for example, chi-square or Mann-Whitney U), and confidence intervals are reported.

F. Recommended Practices

Based on the results, the study derives recommendations such as security-aware prompting, mandatory code review, SAST in CI pipelines, secret scanning, and developer training.

VIII. PROPOSED EVALUATION FRAMEWORK

Stage	Function
1. Task Definition	Define specifications for security-sensitive programming tasks.
2. Code Generation	Produce AI solutions (multiple assistants, prompts, repetitions) and human solutions.
3. Preservation	Store unedited outputs with metadata and SHA-256 hashes.
4. Automated Analysis	Run SAST and secret-scanning tools.
5. Manual Verification	Confirm or reject each finding; apply the checklist.
6. Comparison	Compute metrics and statistical tests across groups.
7. Reporting	Document methods, findings, limitations, and recommendations.

Table II. Seven-stage evaluation framework

IX. SYSTEM ARCHITECTURE

Task specifications → Prompt sets → AI assistants / Human developers → Source code repository (hashed) → SAST tools + secret scanner → Findings database → Manual verification → Metrics and statistics → Research report.

Keeping raw generated code separate from analysed copies improves reproducibility and prevents accidental modification of the original outputs.

X. LEGAL AND ETHICAL CONSIDERATIONS

Vulnerable code produced in this study is intended only for analysis in an isolated environment and must not be deployed. Working exploit development is outside the scope; vulnerabilities are identified and classified, not weaponised. Terms of service of AI assistants should be reviewed before use, and generated code should not be assumed to be free of licensing concerns. Human participants should give informed consent, and their identities and any personal data must be protected in line with applicable data protection law, including India's Digital Personal Data Protection Act, 2023 [7]. Results should be reported fairly, without claiming that any specific product is "insecure" beyond what the data supports.

XI. EXPECTED RESULTS AND DISCUSSION

No numerical results are claimed before the experiment is carried out. The final report should present the number of programs, tasks, languages, and tools; vulnerability counts per category for each group; vulnerable-program rate and density with confidence intervals; scanner precision and disagreement between tools; and the effect of security-aware prompting.

The discussion should interpret which categories are most affected and why, for example whether generated code tends toward string-built SQL queries or hardcoded example credentials because such patterns are common in training data and tutorials. It should also report where AI code performed as well as or better than human code, since the direction of the difference is an empirical question.

XII. LIMITATIONS

- Results depend on specific model versions, which change frequently.
- Non-deterministic outputs mean results may vary between runs.
- Small programs may not represent large real-world systems.
- Human baseline quality depends on participants' experience and may not reflect professional developers.
- SAST tools miss logic flaws and produce false positives; manual review is subjective.
- Findings may not generalise across languages or frameworks not studied.

XIII. FUTURE SCOPE

Future work can include larger, more diverse task sets; additional languages; dynamic testing (DAST) and fuzzing; agentic coding tools that generate multi-file projects; studies of developers actually using assistants; evaluation of automated generate-then-fix loops where the model repairs its own vulnerabilities; and benchmark datasets for reproducible comparison.

XIV. CONCLUSION

AI coding assistants improve productivity, but functional code is not necessarily secure code. This paper proposed a controlled, reproducible framework for comparing LLM-generated and human-written programs across five common weakness classes, combining static analysis with manual verification and standard taxonomies. The central contribution is a methodology that treats AI output as untrusted code, measures its security with transparent metrics, and reports tool errors honestly. Empirical execution of the study will determine the actual vulnerability patterns and support evidence-based guidance for secure AI-assisted development.

REFERENCES

[1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in Proc. IEEE Symposium on Security and Privacy (S&P), 2022.

[2] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," in Proc. ACM Conference on Computer and Communications Security (CCS), 2023.

[3] M. Souppaya, K. Scarfone, and D. Dodson, "Secure Software Development Framework (SSDF) Version 1.1," NIST Special Publication 800-218, 2022.

[4] OWASP Foundation, "OWASP Top 10 – 2021: The Ten Most Critical Web Application Security Risks," 2021.

[5] MITRE Corporation, "Common Weakness Enumeration (CWE)," https://cwe.mitre.org.

[6] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[7] Government of India, "The Digital Personal Data Protection Act, 2023," Act No. 22 of 2023, Aug. 11, 2023.