Research on Security Risk Analysis and Coping Strategies of AI Generated Code

Yanwen Zhang
School of Information Technology and Engineering, Guangzhou College of Commerce, Guangzhou, Guangdong, China

Abstract: The widespread adoption of AI coding assistants such as ChatGPT and GitHub Copilot has transformed software development. However, the code they generate frequently harbors critical security vulnerabilities including SQL injection, cross-site scripting (XSS), and buffer overflow. This paper systematically reviews recent research and industry reports on the security risks of AI-generated code, and proposes a four-dimensional analytical framework encompassing code vulnerabilities, code quality, supply chain security, and developer cognition. Analysis reveals significant interactions among these dimensions: developer cognitive biases may exacerbate vulnerability risks, supply chain contamination degrades code quality, while robust security practices can partially mitigate these adverse impacts. Accordingly, this paper proposes multi-layered mitigation strategies covering technical detection, process management, developer training, and organizational governance, providing actionable guidance for secure software development in the AI-assisted paradigm.

Keywords: AI Coding Assistants; Code Security; Large Language Models; Vulnerability Detection; Software Security

1. Introduction

In recent years, AI coding assistants powered by large language models (LLMs) have advanced rapidly, profoundly reshaping software development workflows. According to Stack Overflow's 2025 Developer Survey [1], ChatGPT and GitHub Copilot have become the mainstream AI-assisted coding tools, Over 80% of surveyed developers use such tools in their daily work. In China, domestic AI coding tools such as Qoder CN and Doubao MarsCode are also gaining rapid traction, and AI-assisted programming is becoming the new norm in software development.

However, while AI coding assistants have significantly boosted development efficiency, they also pose non-negligible security risks. Numerous studies have shown that AI-generated code contains a significant proportion of security vulnerabilities. This issue has attracted widespread attention from both the academic and industrial communities. The Center for Security and Emerging Technology (CSET) at Georgetown University has specifically published a systematic assessment report on the cybersecurity risks posed by AI-generated code [2]. The Open Web Application Security Project (OWASP) has also included AI-related security risks in its Top Ten Risks for Large Language Model Applications [3]. Against this backdrop, accurately identifying the security risks posed by AI-generated code and implementing effective mitigation strategies has become a critical challenge for software security professionals and researchers.

The purpose of this study is to systematically examine the security risks of code generated by AI coding assistants, and to build an analytical framework covering four dimensions: code vulnerabilities, code quality, supply chain security, and developer cognition --and to propose a multi-level response to it. This framework synthesizes existing literature and draws on widely recognized standards such as the NIST Secure Software Development Framework (SSDF) [4] and the NIST AI Risk Management Framework (AI RMF) [5]. The reason why these four dimensions are chosen is that the current research tends to discuss technical vulnerabilities, human–computer interaction or supply chain issues respectively. This paper attempts to integrate these perspectives, and identify the important interdependencies between the dimensions, and then provides a holistic analysis framework for understanding the security of AI code in real scenarios. Grasping these cross-dimensional

Copyright @ STEMM Institute Press

associations is of key significance for the development of practical and effective countermeasures, and the multi-level strategy proposed in this paper is precisely to provide practitioners with operational guidance.

2. Related Work

Existing literature reviews and empirical studies have examined the intersection between AI code generation and software security from diverse perspectives. Negri-Ribalta et al. [6] examined the role of AI-generated code on many levels of software and system security, and pointed out that structured verification is indispensable. Chen et al. [7] reviewed 68 literatures focusing on the security risks of code language models, covering training data poisoning, backdoor attacks, and adversarial perturbations. Hou et al. [8] discussed the application of LLM throughout the entire software development process. Their analysis of the quality and reliability of the generated code lays an important foundation for understanding the security consequences. They list the generation of security vulnerabilities as one of the core problems of LLM-based code generation, alongside the issues of hallucination and inconsistency.

Compared with the existing reviews, the uniqueness of this paper is reflected in the following aspects:

(1) The analytical architecture for practitioners. This paper is based on a four-dimensional framework that places developer cognitive biases on an equal footing with technical deficiencies, a perspective that often does not receive enough attention in reviews that focus on engineering technology.

(2) The frontier of empirical support. This paper introduces recent industry statistics, such as the Veracode's 2025 GenAI Code Security Report [9], to give an empirical reference close to the current situation.

(3) The underlying technical context. This paper discusses the code-specific LLM infrastructure that supports existing AI coding assistants, making up for the technical groundwork that is usually lacking in general NLP reviews.

(4) The countermeasures that can be landed. This paper gives a multi-level response plan for the implementation of reference, and adapts according to the difference of organizational maturity to narrow the gap between academic achievements and industrial applications.

3. Background: AI Coding Assistants and Code Generation

Currently, AI coding assistants fall into three categories: (1) code-completion tools (such as GitHub Copilot), which leverage OpenAI Codex to provide context-aware suggestions; (2) general-purpose conversational AIs (such as ChatGPT and Claude), which can generate complete code snippets based on natural language prompts; (3) AI integrated development environments (such as Cursor and Qoder CN) deeply embed large language models into the editor, providing end-to-end programming assistance. Although these tools differ in their interaction paradigms, their core technology all relies on the code-generation capabilities of large language models.

All tools are based on the Transformer architecture and its variants (CodeBERT, CodeT5, InCoder, AlphaCode), learning statistical patterns from vast amounts of open-source code and generating output token by token in an autoregressive manner. This generative process is inherently probabilistic without a deep semantic understanding of code security, insecure coding patterns in the training data are easily reproduced by the model [10]. Typical applications, such as code completion, function generation, code refactoring, and test case generation, enable AI-assisted code to be deployed directly into production, making its security a critical factor that directly impacts the overall security posture of the final product.

4. Security Risk Analysis of AI-Generated Code

4.1 Code Vulnerability Risk

Code vulnerabilities are the most prominent security risk associated with AI-generated code. Table 1 summarizes the core findings of major studies conducted in recent years.

As shown in Table 1, various studies have confirmed the security vulnerabilities in AI-generated code from multiple perspectives. In a pioneering study, Pearce et al. [11] conducted a systematic evaluation of GitHub Copilot by constructing 89 security-related programming scenarios and found that in approximately 40% of these scenarios, Copilot generated code containing known vulnerabilities. Perry et al. [12] conducted a controlled experiment with 47 participants and found that code written by developers using AI assistants exhibited

 Copyright @ STEMM Institute Press

significantly lower security than that produced by the control group without such assistance, and users tended to be overly confident in their code's security. Fu et al. [4] extended their research from experimental settings to real-world open-source projects, founding that 27.3% of the code generated by Copilot and submitted to GitHub projects is associated with 43 distinct CWE vulnerability types, with C++ exhibiting the highest vulnerability rate.

Table 1. Summary of Major Research Findings on the Security of AI-Generated Code

Research	Institution	sample	Vulnerability rate	Key Findings
Pearce et al. [11]	New York University	89 scenes	40%	Approximately 40% of Copilot-generated code contains security vulnerabilities.
Perry et al. [12]	Stanford University	47 users	Significant difference	Users who employ AI assistants are more likely to write insecure code and may harbor a false sense of security.
Fu et al. [13]	Wuhan University	GitHub project	27.3%	It involves 43 CWE types, with C++ having the highest vulnerability rate.
CSET Report [2]	Center for Security and Emerging Technologies at Georgetown University	Comprehensive Assessment	systematic	AI-generated code is vulnerable to a range of risks, including adversarial attacks and data poisoning.

4.2 Common Vulnerability Types

According to MITRE's CWE classification scheme [14], security vulnerabilities in AI-generated code can be categorized as follows:

Injection vulnerability (CWE-89/CWE-79). SQL injection and cross-site scripting (XSS) are two of the most common injection vulnerabilities. When generating database queries or web frontend code, AI models often resort to string concatenation instead of parameterized queries, thereby introducing the risk of SQL injection. Similarly, failure to properly escape user input when generating HTML output can introduce XSS vulnerabilities.

Insecure cryptographic practices (CWE-327/CWE-328). AI models may employ cryptographic algorithms that have been proven insecure, such as using MD5 or SHA-1 for password hashing, or resort to hardcoding in key management. Torka and Albayrak [7] pointed out that AI-generated cryptography-related code suffers from issues such as the use of outdated APIs and weak cryptographic algorithms.

Authentication and Authorization Vulnerabilities (CWE-287/CWE-862). AI-generated authentication codes may lack essential security safeguards, such as improper session management and missing permission checks. These vulnerabilities are particularly common in the backend code of web applications.

Sensitive information disclosure (CWE-200/CWE-522). AI models may embed hardcoded API keys, database credentials, or internal URLs in the code they generate, or expose sensitive data in log outputs.

Improper resource management (CWE-400/CWE-770). AI-generated code may lack proper resource-release logic, such as unclosed database connections or unreleased file handles, leading to resource leaks and denial-of-service risks.

4.3 Vulnerability Distribution Characteristics and Language Differences

Drawing on the research data of Fu et al. [13] and Pearce et al. [11] the distribution of security vulnerabilities in AI-generated code exhibits a pronounced pattern of concentration. Table 2 lists the vulnerability types with the highest frequency of occurrence.

The data in Table 2 shows that the security vulnerabilities in AI-generated code are primarily concentrated in three categories: insufficient input validation, improper cryptographic practices, and resource management defects. The above issues are highlighted in MITRE's CWE Top 25 Most Dangerous Software Weaknesses list [14], which indicates the insufficient representation of the security coding paradigm in the AI training data.

Fu et al. [13] also found that AI code risks vary by programming language: the vulnerabilities rate from high to low is C++, Python and Java. This distribution is closely related to the security properties of each language itself. C++ permits low-level memory operations, making it susceptible to buffer overflows and memory safety vulnerabilities. Although Python has memory management protection, its dynamic type and interpretation execution method make the generated code more vulnerable to injection

Copyright @ STEMM Institute Press

attacks in the absence of input verification. Java relies on the strong type system to provide some protection, but if the framework is called improperly, it will also introduce risks. Accordingly, when developers use AI coding assistants, differentiated security reviews should be adopted for languages: C/C++ focuses on memory security checks, Python focuses on input validation checks, and Java focuses on framework usage audits.

Table 2. Common Types of Security Vulnerabilities in AI-Generated Code (Based on the CWE Classification)

CWE number	Vulnerability Name	Typical manifestations in AI-generated code
CWE-89	SQL injection	Constructing SQL queries using string concatenation without parameterized queries.
CWE-79	Cross-Site Scripting (XSS)	HTML escape not performed when outputting user input.
CWE-120	Buffer overflow	Use of unsafe string manipulation functions, lacking boundary checks
CWE-327	Using a cryptosystem that has been broken	Using weak algorithms such as MD5 and DES for password hashing or data encryption
CWE-200	Sensitive information leakage	Hardcoding API keys, outputting sensitive data in logs
CWE-287	Improper identity authentication	Missing session validation; permission-checking logic is incomplete.
CWE-400	Improper resource management	Database connection not closed, file handle not released.
CWE-862	Authorization check missing	The API interface has not validated the caller's permissions.

4.4 Code Quality and Code Smell Issues

Beyond direct security vulnerabilities, the code generated by AI also exposes quite common quality shortcomings. Liu et al. [15] carried out a multi-dimensional evaluation of the code generated by ChatGPT, and the results pointed out that it was deficient in four aspects: functional correctness, code complexity, maintainability, and security. Typical problems include: redundant code structure, excessively long functions, imperfect exception handling mechanism, non-standard naming conventions, etc. Gasiba et al. [16] further confirmed through systematic testing that ChatGPT often generates code that appears to be operational but in fact deviates from established security coding specifications, such as failing to properly purify the input or not using a secure random number generator. Although such quality issues may not directly evolve into security vulnerabilities, they can make it more difficult to review and increase maintenance expenditures. Not only that, poor-quality code may also obscure deeper defects; overly complex functions and lack of consistency in structure can weaken the effectiveness of manual security audits and automated static analysis, thus creating conditions that allow subtle defects to evade detection.

4.5 Supply Chain Security Risks

AI coding assistants have also introduced supply chain security risks that are relatively uncommon in traditional software development. The CSET assessment report [2] points out that AI models may generate code that references malicious software packages, employ non-existent dependency libraries, or replicate contaminated code snippets from the training data. Spracklen et al. [17] conducted a comprehensive analysis of package hallucinations across multiple code-generating LLMs. This demonstrates that illusion packages can be exploited by attackers, who can register these fictitious package names in public repositories, thereby creating a novel attack vector. Furthermore, when processing code, AI coding assistants must transmit the developer's code context to cloud servers, which process inherently carries risks of source-code leakage and sensitive information exposure. OWASP's Top Ten Security Risks for LLM Applications [3] also identifies supply-chain vulnerabilities, training-data poisoning, and sensitive information leakage as core security threats to LLM applications.

4.6 Developer Cognitive Bias

AI coding assistants can also give rise to cognitive biases among developers. Perry et al. [12] found in their experiment that developers using AI assistants not only produced more insecure code but also exhibited higher subjective self-assessments of their code's security, demonstrating overconfidence in its safety. This bias stems from developers' tendency to regard AI-generated code as "high-quality output that has been trained", which lowers vigilance and reduces the resources allocated to reviewing AI-generated code.

5. Response Strategies and Best Practices

5.1 Static Analysis and Automated Detection

The first line of defense against AI-generated code security risks is automated detection tools. Static application security testing (SAST) tools

 Copyright @ STEMM Institute Press

can identify potential security vulnerabilities without executing the code, making them well-suited for conducting large-scale security reviews of AI-generated code. Current mainstream SAST tools include CodeQL (developed by GitHub, supports multilingual semantic analysis), SonarQube (a comprehensive tool for assessing code quality and security), and Semgrep (a lightweight, rule-based code scanning tool). Snyk's security report [18] recommends integrating SAST scans as automated pre-commit hooks into development workflows that utilize AI coding assistants, ensuring that AI-generated code undergoes security screening before being integrated into the codebase. In addition, a number of security-oriented tools specifically designed to detect AI-generated code are also emerging. For example, GitHub Copilot itself has integrated CodeQL-based security scanning, which can filter out obviously insecure code suggestions during the code-completion process.

To demonstrate the effectiveness of using automated reviews in conjunction with secure coding specifications, a typical example is given below. If the following request is made to ChatGPT (GPT-4): "Please write a function in Python to retrieve user data from the database based on user input", its return code constructs the SQL statement using direct string concatenation, which constitutes a classic SQL injection flaw (CWE-89). Submit this output to Semgrep configured with the OWASP Top Ten rule set; it immediately flags the vulnerable line with a high-severity alert. The model is then prompted again with the explicit instruction to "Use parameterized queries to prevent SQL injection". The newly generated code changes to the appropriate parameter binding method and passes all Semgrep checks smoothly. This case is not complex, but it illustrates two points: (a) the automated SAST tools can effectively identify common vulnerability patterns in AI-generated outputs; (b) explicit security instructions can significantly improve the quality of the generated results.

5.2 Integrating Security into the Development Workflow

Relying solely on technical tools is insufficient to comprehensively address AI-code security risks; systematic integration at the development-process level is also required. The specific recommendations are as follows:

Mandatory human review of AI-generated code. All code generated by or with the assistance of AI shall be subject to the code review process, and reviewers must maintain a security-aware mindset, with particular focus on the high-frequency vulnerability types listed in Table 2. Perry et al. [12] emphasize that manual review is an indispensable step for addressing security vulnerabilities in AI-generated code.

Adaptation of the secure coding guidelines. Organizations should develop secure coding guidelines tailored to AI-assisted programming scenarios, clearly defining the baseline security requirements for using AI coding assistants. For example: the direct use of AI-generated code related to authentication and cryptography without prior review is prohibited; all database operations must employ parameterized queries; and API endpoints are required to incorporate access control logic.

Defense-in-depth strategy. At the software architecture level, security design patterns such as the principle of least privilege, input validation, output encoding, and defense in depth should be adopted to ensure that even if individual security vulnerabilities exist in AI-generated code, they will not result in system-level security incidents.

It should be noted that the manner in which the above countermeasures are implemented will vary depending on the organizational context. Each organization needs to strike a balance between development efficiency and the rigor of security, and select the appropriate solution based on its own development scale, risk tolerance level and current security capability maturity. For example, smaller teams can prioritize automated SAST scanning and mandatory code reviews. Larger institutions are better placed to invest in model fine-tuning and build more comprehensive governance systems. At the same time, in view of the changing level of AI technology and threat environment, relevant strategies need to be reassessed and updated on schedule.

5.3 Enhancing Developers' Security Awareness

With the increasing popularity of AI coding assistants, the security literacy required by developers has also increased. First, they should be able to identify potential risks embedded in AI-generated code, which requires a solid foundation in secure coding practices. Secondly,

Copyright @ STEMM Institute Press

developers must adopt the correct perspective on AI tools: viewing them as assistants that enhance efficiency rather than "security experts", always maintain the habit of rigorously verifying the model's outputs. To this end, organizations should establish a regular AI code security training program, leveraging case studies and hands-on exercises to enhance developers' ability to identify AI code vulnerabilities.

5.4 Security Governance Framework at the Organizational Level

In the organizational governance dimension, this paper proposes the development of an AI programming security governance framework that encompasses the following key elements:

(1) Establish guidelines for the use of AI programming tools. Clearly specify which development tasks can be assisted by AI, and restrictions should be imposed on sensitive code categories such as authentication modules, cryptographic implementations and access control logic: such code must not be directly generated by AI without mandatory expert review.

(2) Establish an AI-based code security audit system. Conduct periodic specialized reviews of AI-generated code, with a focus on identifying the high-frequency vulnerabilities listed in Table 2 and combine automated scanning with manual review.

(3) Introduce traceability and tracking mechanism. Leverage version control and annotation to mark AI-generated code segments, enabling secure post-review and vulnerability localization. This approach also supports organizations in conducting retrospective assessments of the temporal evolution of AI code security impacts.

(4) Establish a security incident response process tailored to AI programming activities. Organizations must develop dedicated response procedures for security incidents arising from AI-related programming, ensuring that any issues can be promptly identified and remediated.

6. Empirical Validation: Controlled Experiments on Cognitive Biases

To validate the cognitive bias dimension (Dimension 4) within the proposed framework, this study conducted a controlled experiment to examine how the information that "the code was generated by AI" influences developers' vulnerability detection behavior and their perception of security.

6.1 Experimental Design

Eighty-two junior and senior undergraduate students majoring in Software Engineering and Computer Science and Technology at Guangzhou College of Commerce participated on a voluntary basis. All participants have completed at least one course in web development or database programming. The study received ethical approval and obtained informed consent from all participants.

Participants were randomly assigned to two groups, with 41 participants in each group. The code materials reviewed by both groups were identical. Group A was informed that the code snippets had been "generated by ChatGPT based on user prompts," while Group B was told that these snippets had been "collected from publicly available open-source repositories on GitHub." Five code snippets (three in Python, two in Java, each containing 30 to 60 lines of code) were generated using ChatGPT and subjected to a preliminary evaluation by two independent security experts to ensure consistent difficulty. Each code snippet contains two known security vulnerabilities, selected from the high-frequency types identified in Table 2: SQL injection (2), XSS (2), weak encryption (1), hard-coded credentials (2), missing authorization (2), and improper input validation (1).

Participants completed a paper-based questionnaire within 45 minutes, which included: (1) reviewing each excerpt and annotating any identified safety concerns; (2) rating the overall safety of each excerpt on a 5-point Likert scale (1 = "very unsafe," 5 = "very safe"); and (3) providing statistical information, including programming experience and previous use of AI tools. The use of automated analysis tools or the internet is prohibited during the experiment.

6.2 Experimental Results

Table 3 summarizes the key performance metrics for vulnerability detection.

Table 3. Vulnerability Detection Results by Experimental Group

Measure	Group A (AI, n=41)	Group B (OSS, n=41)	t-statistic	p-value
Mean detected (of 10)	3.41 (SD=1.62)	4.73 (SD=1.84)	t(80) = -3.46	< 0.001
Detection rate	34.1%	47.3%		

 Copyright @ STEMM Institute Press

Mean security rating	3.56 (SD=0.68)	2.83 (SD=0.74)	U = 512.5	< 0.01

The number of vulnerabilities detected in Group A was significantly lower than that in Group B (Group A: M = 3.41, SD = 1.62; Group B: M = 4.73, SD = 1.84). An independent-samples t-test confirmed the statistical significance (t (80) = −3.46, p <.001, Cohen's d = 0.76, indicating a medium-to-large effect size). A difference of 13.2 percentage points in the overall detection rate indicates that knowing the code was generated by AI significantly reduces participants' critical evaluation of it.

Detection gaps vary depending on the type of vulnerability, as shown in Table 4.

Table 4. Per-Instance Vulnerability Detection Rates by Type and Group

Vulnerability Type	Group A (AI)	Group B (OSS)	Difference
SQL Injection (CWE-89)	39.0%	56.1%	-17.1 pp
XSS (CWE-79)	31.7%	46.3%	-14.6 pp
Weak Cryptography (CWE-327)	29.3%	51.2%	-22.0 pp
Hardcoded Credentials (CWE-522)	40.2%	46.3%	-6.1 pp
Missing Authorization (CWE-862)	24.4%	34.1%	-9.8 pp
Improper Input Validation (CWE-20)	41.5%	56.1%	-14.6 pp

Note: Detection rate is the proportion of vulnerability instances detected within each type (e.g., for a type with 2 instances across 5 snippets, the denominator is 2x41=82). Weighted sums of per-type rates (by instance count) equal the overall means reported in Table 3.

It is worth noting that for defects with more salient characteristics, such as hard-coded credentials, the inter-group difference is only 6.1 percentage points, the smallest among all cases, indicating that overt flaws are generally easier to detect regardless of their underlying causes. In contrast, vulnerabilities that are more subtle, particularly weak encryption flaws that require a deep understanding of security semantics, exhibit inter-group differences as high as 22.0 percentage points, which is the largest.

Regarding safety perception, Group A's ratings of the code snippets on a 5-point Likert scale were significantly higher than those of Group B (Group A: M=3.56, SD=0.68; Group B: M=2.83, SD=0.74). The Mann-Whitney U test confirmed this difference (U=512.5, p <0.01). Crucially, among all participants, the number of vulnerabilities detected was negatively correlated with the average security score (r=−0.42, p<0.001): participants who identified fewer vulnerabilities rated the code as more secure. This negative correlation is consistent with the "false sense of security" mechanism proposed by Perry et al. [3]: although the same materials are being reviewed, participants who were informed that the code was generated by AI both detected fewer defects and rated the code as more secure.

To rule out the influence of other factors, this experiment further examined the relevant statistical variables. The results showed that participants who had used AI tools (63.4% of the total sample) did not exhibit any significant differences in detection rates (p=0.34, Cohen's d=0.21) or safety scores (p=0.51, Cohen's d=0.14) compared to those with no prior experience. Similarly, self-reported safety knowledge (on a 5-point scale) was also not a significant predictor of performance (p=0.22, β=0.15). The aforementioned evidences collectively indicate that the illusory-safety effect does not appear to depend on individuals' prior experience with AI or their self-assessment of their own abilities, thereby providing stronger support for a cognitive bias explanation.

6.3 Implications for the Four-Dimensional Framework

The results of this experiment directly support Dimension 4 in the framework, namely developer cognitive bias. Under AI attribution conditions, the vulnerability detection rate decreased by 27.9%, while the security score was inflated by 25.8%. These results confirm that "false sense of security" exhibits cross-dimensional interactions. Additionally, vulnerability categories exhibit differentiated impacts: the higher the degree of a defect's concealment the greater the difference in the test results. This indicates that cognitive biases selectively undermine vulnerability detection tasks that require deeper code understanding, while obvious defects remain unaffected by attribution conditions and can still be detected.

The aforementioned findings also hold significant implications for practical applications. Since the illusory-safety effect persists across different levels of experience, this implies that the training interventions described in Section 5.3 must directly address the cognitive bias dimension, rather than assuming that enhancing technical competence alone will suffice to curb the uncritical acceptance of AI-generated outputs. The mandatory manual review process (Section 5.2) requires the configuration of detailed verification checklists,

Copyright @ STEMM Institute Press

with a particular focus on vulnerability categories most susceptible to cognitive biases, such as SQL injection and XSS.

7. Challenges and Future Directions

Although the aforementioned strategies can mitigate the security risks of AI-generated code to some extent, the field still faces numerous challenges.

First, the opacity of models—often referred to as the "black-box" problem—means that risks cannot be completely eliminated. Although LLMs are continuously being iterated, they still lack a deep understanding of the security semantics of code and cannot provide interpretable safety guarantees during generation. Even with the aid of safety alignment techniques to constrain the output of malicious code, it is also difficult to cover all possible security vulnerability patterns. At the same time, this lack of transparency also makes debugging and patching more challenging after vulnerabilities are exposed.

Secondly, as the functionality of AI coding assistants continues to expand, their associated security risks are also evolving dynamically. From early code completion to the emergence of autonomous agent-based programming tools such as Devin and Cursor's Agent Mode, AI coding assistants are progressively taking on core roles in architectural design, test case generation, and deployment configuration. As a result, risks extend along two axes: horizontally, they affect more stages in the software development process; vertically, they involve more complex system-level vulnerabilities that are increasingly difficult to identify and trace.

Third, security detection technologies for AI-generated code are still in their early stages.

Most existing SAST tools are designed to detect common vulnerability patterns in manually written code. However, their ability to identify AI-specific security challenges, such as hallucination-induced vulnerabilities, latent defects caused by contaminated training data, and logical errors arising from statistical pattern matching，remains limited.

In the future, AI code security research can advance along several pathways: first, establish specialized security testing benchmarks and evaluation frameworks tailored to AI-generated code; second, explore LLM-based secure code-generation strategies that enable programming assistants to proactively adhere to security guidelines during the generation process, including safety-aware fine-tuning and reinforcement learning based on human feedback; third, exploring a new paradigm for secure human–AI collaboration in programming that leverages the efficiency advantages of AI while ensuring code security through a well-defined division of labor—for example, using AI to generate boilerplate code and reserving the implementation of safety-critical components for human developers, thereby optimizing human–AI teamwork.

8. Conclusion

AI coding assistants are continuously transforming software development practices, but the security risks posed by the code they generate must be given due attention. Building on a systematic review of recent authoritative international research, this paper analyzes the core security threats posed by AI-generated code across four dimensions: code vulnerabilities, code quality, supply chain security, and developer cognition. Furthermore, a comprehensive response strategy has been developed, integrating automated detection, process management, personnel training, and organizational governance.

Research indicates that a high proportion of code generated by AI contains security vulnerabilities, including SQL injection, cross-site scripting, and weak encryption, among others, and the risk levels vary across different programming languages. A more insidious risk is that AI tools may instill a "false sense of security" in users, thereby diminishing their vigilance toward code vulnerabilities. To address the aforementioned challenges, it is essential to integrate technical measures with management systems, thereby harnessing the efficiency gains of AI-driven programming while establishing a multi-layered security assurance framework.

References

[1] Stack Overflow. 2025 Developer Survey: AI. https://survey.stackoverflow.co/2025/ai, 2025.

[2] Ji J, Jun J, Wu M, et al. Cybersecurity risks of AI-generated code. Center for Security and Emerging Technology (CSET), Georgetown University, 2024. https://cset.georgetown.edu/publication/cybersecurity-risks-of-ai-generated-code/.

[3] OWASP Foundation. OWASP Top 10 for

 Copyright @ STEMM Institute Press

LLM Applications 2025. (2024-11-18)[2026-08-13]. https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/.

[4] National Institute of Standards and Technology. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication 800-218, 2022.

[5] National Institute of Standards and Technology. Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1, 2023.

[6] Negri-Ribalta C, Geraud-Stewart R, Lenzini G. A systematic literature review on the impact of AI models on the security of code generation. Frontiers in Big Data, 2024, 7: 1386720.

[7] Chen Y, Sun W, Fang C, et al. Security of language models for code: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 2025, 35(7): 1-42.

[8] Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 2024, 33(8): 220:1-220:79.

[9] Veracode. 2025 GenAI code security report. Veracode, 2025.

[10] Hendrycks D, Basart S, Kadavath S, et al. Measuring coding challenge competence with APPS. Advances in Neural Information Processing Systems (NeurIPS), 2021, 34: 24018-24030.

[11] Pearce H, Ahmad B, Tan B, et al. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022: 980-997.

[12] Perry N, Srivastava M, Kumar D, et al. Do users write more insecure code with AI assistants?. Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23). ACM, 2023: 2255-2269.

[13] Fu Y, Liang P, Tahir A, et al. Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. ACM Transactions on Software Engineering and Methodology, 2025, 34(1): 1-31.

[14] MITRE. CWE Top 25 Most Dangerous Software Weaknesses 2025. (2025-12-16)[2026-08-13]. https://cwe.mitre.org/top25/.

[15] Liu Z, Tang Y, Luo X, et al. No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. IEEE Transactions on Software Engineering, 2024, 50(6): 1504-1538.

[16] Gasiba T, Iosif A-C, Kessba I, et al. May the source be with you: On ChatGPT, cybersecurity, and secure coding. Information, 2024, 15(9): 572.

[17] Spracklen J, Jadliwala M S, Raval A, et al. We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. Proceedings of the 34th USENIX Security Symposium (USENIX Security '25). USENIX Association, 2025.

[18] Snyk. State of AI-generated code security. Snyk, 2023. https://snyk.io/state-of-ai-generated-code-security/.

Copyright @ STEMM Institute Press