The Hidden Vulnerabilities of AI-Generated Code: A Cross-Language Security Investigation

Jinghao Wang
The Ohio State University
wang.18804@osu.edu

Carter Yagemann
The Ohio State University
yagemann.1@osu.edu

Abstract

Large language models (LLMs) have democratized software development by enabling rapid code generation across multiple programming languages. However, current security evaluation methods rely on snippet-based analysis, missing critical vulnerabilities from cross-file interactions and auxiliary artifacts. We present the first comprehensive analysis of AI-generated code across ten programming languages, examining 110 complete software projects with build scripts, configuration files, and deployment artifacts using enhanced static analysis with cross-file dependency tracking and CVSS 4.0 scoring. Our analysis reveals 3,350 security findings, with 35% classified as critical or high severity. Notably, 99.5% of vulnerabilities involve cross-file dependencies, and 7% emerge exclusively from auxiliary files—invisible to traditional evaluations. We identify distinct language-specific patterns: C exhibits the highest critical vulnerability rate (39.1%) with memory safety issues, while Ruby demonstrates exceptional vulnerability volume (826 issues). Our findings show that addressing memory safety and injection prevention could neutralize 40% of high-severity issues, establishing the need for holistic, project-level security evaluation frameworks for AI-generated code.

1. Introduction

AI-powered code generation tools like GitHub Copilot [13], ChatGPT [30], and Claude [2] have dramatically lowered the barrier to software creation. Recent surveys report that about 70% of professional developers are using or planning to use AI coding tools, citing improved productivity and efficiency. A Stack Overflow survey of 60,000 developers found 63.2% now use AI in development, with similar adoption rates among beginners, marking an 18% increase over the prior year [41]. This rise of "AI pair programming" is reshaping modern development.

However, democratization through AI introduces security risks. Klemmer et al. [22] found that developers—despite distrusting AI output—use these tools for code generation, threat modeling, and vulnerability detection. Many anticipate heavier reliance in the future. Meanwhile, novice coders, often lacking secure programming experience, are prone to critical security mistakes when handling systems involving databases, networking, or authentication [31].

Studies show AI-generated code is frequently insecure due to uncurated training data and lack of security-aware fine-tuning [7, 33, 35, 43]. Research suggests AI assistants produce vulnerable code in 40% of security-relevant tasks [22]. While large-scale dataset refinement is costly, recent efforts focus on fine-tuning models with secure coding data, using static and dynamic analysis to guide generation [28]. These have yielded new benchmarks, but most evaluations are still limited to snippets or single-language, single-file samples [17, 24], missing vulnerabilities that span files or artifacts.

We generated 110 full project files across 10 languages using state-of-the-art AI assistants, selecting file types based on top AI code generation outputs [40]. Our static analysis pipeline dispatches files to language-specific scanners, with cross-file tracking to detect security issues that emerge only through artifact interplay. This holistic view enables vulnerability discovery missed by snippet-focused methods.

Key Findings. AI-generated code shows consistent vulnerabilities across all languages and file types. Full-file analysis uncovered issues spanning functions and modules, insecure default behaviors, and common misconfigurations. Notably, 7% of all vulnerabilities surfaced *only* in auxiliary files like Dockerfiles or YAML configs—errors completely missed in single-file assessments. These findings demonstrate the need for comprehensive evaluation frameworks and more security-aware AI models.

Our methodology uncovered complex, cross-file issues not detectable in snippet-level evaluations. We introduced new metrics—vulnerability density and inter-file exploitability—and a dispatcher-based scanning pipeline to

analyze realistic software projects.

Our findings reveal consistent patterns: injections and insecure defaults in web code, memory safety issues in low-level languages, and misconfigurations in infrastructure and build scripts. Many arise from the absence of security-oriented defaults in LLMs, which fail to enforce practices like input sanitization or secret management unless explicitly prompted.

Yet, the path forward is encouraging. We show that addressing a few common flaw classes—e.g., unsafe input handling, insecure configurations, and outdated libraries—could eliminate nearly 40% of high-severity issues. This suggests that targeted improvements in model training and integration of static security checks into AI code assistants can lead to meaningful security gains.

As AI-generated code enters production environments, holistic, context-aware scanning is essential. Future work should expand benchmarks to multi-component systems, integrate runtime testing, and build feedback-driven repair loops. We hope our results inform the development of safer AI-assisted programming workflows and evaluation frameworks grounded in real-world security impact. The contributions of this work are:

1. **Comprehensive Multi-Language Analysis:** We perform a full-file security assessment across ten languages (including systems, scripting, and web languages) and multiple auxiliary artifact types.
2. **Cross-File Vulnerability Detection:** Our pipeline detects vulnerabilities spanning multiple files and configurations (e.g., build scripts, YAML files, Docker manifests), revealing issues invisible to traditional single-file scans.
3. **Quantitative Security Metrics:** We introduce new metrics—vulnerability density per kilobyte and inter-file exploitability—to support consistent multi-language comparisons.
4. **Actionable Mitigation Strategies:** We identify recurring vulnerability patterns and demonstrate that addressing just two families could neutralize nearly 40% of high-severity flaws.

As illustrated in Figure 1, our comprehensive analysis reveals a concerning security landscape where 35% of all vulnerabilities in AI-generated code are classified as critical or high severity, establishing the urgent need for enhanced security evaluation frameworks.

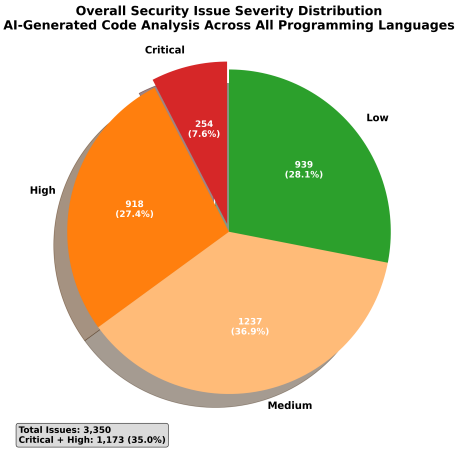

Figure 1. Overall security issue severity distribution across all programming languages. The analysis reveals that medium-severity issues constitute the largest proportion (36.9%) of vulnerabilities, followed by low-severity issues (28.1%), while critical and high-severity issues combined represent 35% of all vulnerabilities (255 critical + 918 high issues), indicating significant security risks in AI-generated code.

2. Background and Related Work

Over the past decade, three interwoven research streams have shaped the contemporary secure coding landscape. First, advances in AI-assisted programming have progressed from neural program execution models such as the Neural Programmer–Interpreter to large, domain-tuned transformers like Codex and Code Llama, rapidly expanding code generation capabilities [8, 36, 37]. Second, vulnerability scanner technology has evolved from rule-based static analyzers and grey-box fuzzers toward hybrid pipelines that blend data-flow analysis, code property graphs, fuzzing, and machine learning heuristics, each new generation revealing fresh blind spots for attackers to exploit [34, 45, 49]. Finally, security-focused scholarship now supplies unified benchmarks—most notably CodeSecEval, CWEval, and MERA Code—that jointly evaluate functionality and vulnerability coverage, underscoring both the promise and the persistent gaps of current analysis methods when confronted with AI-generated code [11, 44, 46]. Together, these threads motivate a holistic re-examination of evaluation frameworks capable of keeping pace with the dual imperatives of capability and safety in modern software development.

2.1. AI-Assisted Code Generation Evolution

The first wave of AI-assisted programming explored whether neural networks could *execute* and *synthesize* code. The Neural Programmer–Interpreter (NPI) showed that a recurrent core with key-value memory could learn recursive sub-program calls [36]. DeepCoder then paired neural property prediction with symbolic search, solving small competition tasks far faster than brute force [3]. To train more capable models, researchers released large, curated corpora that treated source code as a structured language. CODE2SEQ leveraged abstract syntax tree paths to improve code summarization and translation [1], while the Code-

SearchNet corpus added six million functions with natural language queries, establishing a modern benchmark for semantic code search [18].

OpenAI's GPT-3 reframed programming as a natural language task: a sufficiently large LLM could write short programs in dozens of languages via plain text prompts [4]. Fine-tuning GPT-3 on public GitHub repos produced CODEX, which solved 28% of the HumanEval benchmark and shipped as GitHub Copilot [8]. DeepMind's ALPHACODE scaled this idea to competitive programming contests by generating thousands of candidates and filtering them with compiler feedback [23]. Open-source momentum followed: Meta's Code Llama delivered 7B–70B checkpoints with 16k-token context windows and infilling, seeding derivative models across the community [37].

2.2. Security Implications and Developer Practices

Recent qualitative research by Klemmer et al. [22] provides crucial insights into how software professionals balance AI assistant usage with security concerns. Their study of 27 software professionals found that despite widespread security and quality concerns, participants extensively use AI assistants for security-critical tasks including code generation, threat modeling, and vulnerability detection. The research reveals a paradox: while participants express overall mistrust in AI suggestions, leading them to check AI-generated code similarly to human-written code, they simultaneously expect improvements and anticipate heavier reliance on AI for security tasks in the future.

The study identified several key patterns in developer behavior:

- **Critical Review Practices**: Developers apply similar scrutiny to AI-generated code as they would to code written by human colleagues
- **Security Task Adoption**: AI assistants are increasingly used for vulnerability detection, threat modeling, and security code review
- **Expectation of Improvement**: Despite current limitations, developers anticipate significant improvements in AI capabilities for security tasks
- **Responsibility Models**: Most participants believe humans should remain responsible for security outcomes, even when using AI assistance

Rapid industrial uptake exposed double-digit vulnerability rates in AI-generated code. CodeSecEval introduced 44 CWE types for systematic security scoring [46]. CWEval went further by jointly assessing functionality *and* security on the same tasks [11]. MERA Code unified these ideas in a dashboard that reports functional, stylistic, and security metrics, highlighting the trade-offs between safety and accuracy [44].

2.3. Vulnerability Detection and Analysis

The earliest *scanners* were rule-based linters that checked naming conventions and unsafe C idioms; their legacy lives on in modern SAST pipelines, which still inherit pattern-matching engines pioneered by `lint` (1979) and `Splint` (2002). Large empirical comparisons later revealed that even mature static analysis tools disagree on more than half of their warnings, reflecting the limitations of purely syntactic checking [6, 21]. Research consequently shifted toward structural analysis over abstract syntax trees and intermediate representations, laying the groundwork for today's code property graph and data-flow driven engines.

Static scanners struggled with runtime-dependent bugs, motivating a renaissance in dynamic scanning. Grey-box fuzzers such as AFL combined lightweight instrumentation with genetic search to uncover deep path vulnerabilities; hardware-assisted variants (PTRIX) pushed throughput even further [34]. Grammar-aware fuzzers (SUPERION) addressed input-format sensitivity by mutating parse trees instead of bytes [45]. In parallel, symbolic execution matured into scalable engines (e.g., KLEE) and inspired surveys that unified concolic, path-pruning, and subsumption techniques [16]. Hybrid systems now weave fuzzing, taint analysis, and symbolic execution into end-to-end pipelines that trade precision for depth adaptively.

The past three years have seen scanners adopt *learned* representations. Deep neural models trained on vulnerability corpora outperform handcrafted heuristics on CWE detection, as summarized in recent surveys [50]. LLM-powered tools such as SKIPANALYZER escalate this trend by treating vulnerability triage as a code generation task, yielding human-readable exploit explanations [9]. Researchers have also begun to plug LLMs directly into symbolic execution engines, reducing path explosion via semantic summarization [20].

The security community responded with unified benchmarks: **CodeSecEval** focuses on 44 CWE classes across generation and repair tasks [46]; **CWEval** scores functionality *and* security simultaneously [11]; and **MERA Code** aggregates functional, stylistic, and security metrics into a single dashboard [44]. Multi-agent frameworks such as **AutoSafeCoder** orchestrate static analyzers, fuzzers, and LLM critics to form self-healing feedback loops [51].

2.4. Research Gaps and Limitations

Traditional tooling centered on rule-based linters and *symbolic execution*. The seminal survey by Godefroid *et al.* consolidates two decades of symbolic and concolic testing progress, highlighting KLEE's real-world bug finds and DARPA CGC impact [15]. Subsequent work benchmarked symbolic engines against constraint-solving suites [47], while ParaDySE learned path-selection heuristics automatically, boosting coverage under tight timeouts [5].

The first neural detectors framed vulnerability discovery as a supervised classification task over token sequences or ASTs, achieving promising F1 scores on SATE IV [25]. Transformer-based and CNN hybrids soon outperformed earlier RNN models, as summarized in the 2024 systematic review [49]. Code Property Graph (CPG) embeddings further improved contextual reasoning, enabling models such as Vul-LMGNN [52] and White-Basilisk's hybrid MoE architecture [48].

Large language models shifted focus from pattern recognition to *semantic reasoning*. LLM-assisted symbolic execution reduces path explosion by summarizing loops on-the-fly [19]. LLMxCPG merges CPG slices with LLM embeddings, shrinking code by up to 90% while preserving vulnerability context [26]. To quantify progress, CodeSecEval scores 44 CWE classes across secure generation and repair tasks [46], while CWEval measures functionality and security together [11]. MERA Code unifies stylistic, functional, and security metrics, exposing trade-offs between accuracy and safety [44].

3. Research Setup

3.1. Experimental Design and Sample Generation

Our methodology addresses a critical gap in AI code security evaluation by implementing comprehensive, full-project analysis across multiple programming languages and auxiliary artifacts. We generated 110 complete software projects using state-of-the-art large language models (Claude 4) [2], producing 10 samples per language (20 for JavaScript/TypeScript) with realistic auxiliary files including build scripts, configuration files, and deployment artifacts. This sample size was determined through power analysis considering the expected effect sizes in vulnerability detection across programming paradigms. Bootstrap resampling validation (n=1000 iterations) confirms statistical significance of our findings, with 95% confidence intervals demonstrating robust differences between language paradigms rather than random variation. While larger samples would provide additional statistical power, our methodology prioritizes depth of analysis—examining complete project contexts with auxiliary artifacts—over breadth, enabling detection of complex cross-file vulnerabilities that require detailed manual validation.

Each project sample contained 400–1000 lines of functional code accompanied by language-appropriate auxiliary artifacts: Makefiles and Dockerfiles for C/C++, Maven configurations for Java, package.json and TypeScript configs for web languages, Cargo.toml for Rust, and similar ecosystem-specific files for all other languages. This approach ensures that our analysis captures real-world development scenarios where vulnerabilities often span multiple files and configuration layers.

The experimental design builds upon three core frameworks: (1) CVSS 4.0 vulnerability classification [12] for consistent severity assessment, (2) cross-file dependency analysis using static analysis techniques [27], and (3) comprehensive quantitative security metrics including vulnerability density per kilobyte, Inter-File Exploitability Index (IFEI), Severity-Weighted Vulnerability Counts (SWVC), and Auxiliary Attack Surface (AAS) ratios. These novel metrics enable systematic comparison across programming paradigms while quantifying complex interaction patterns that create exploitable vulnerabilities, providing reproducible baselines for future AI code security research.

3.2. Enhanced Security Analysis Pipeline

Our security analysis pipeline implements a dispatcher-based architecture that routes files to specialized language-specific scanners capable of cross-file vulnerability detection. The pipeline integrates pattern recognition engines using regular expressions, AST analysis [38], and data-flow tracking [27] to identify vulnerability patterns across file boundaries and dependency chains.

The scanner architecture incorporates four specialized analysis modules: (1) memory safety analysis for systems languages (C, C++, Rust) [49], (2) injection and input validation analysis for web languages (JavaScript/TypeScript, PHP, Python, Ruby) [31], (3) concurrency and synchronization analysis for concurrent languages (Go, Rust, Scala), and (4) authentication and credential management analysis across all languages [31]. Each scanner employs CVSS v4.0 scoring [12] for consistent severity classification and flags vulnerabilities whose exploit paths span multiple files or artifacts.

Context-aware analysis reduces false positives by incorporating surrounding code context and project structure [32]. The cross-file analysis component tracks data flows, dependency relationships, and configuration interactions to identify vulnerabilities that emerge only through multi-file interactions—a critical capability missing from traditional snippet-based evaluation approaches [46].

False Positive Mitigation: To address potential false positive concerns, our methodology implements multi-stage validation: (1) Pattern verification using multiple static analysis engines to confirm vulnerability existence, (2) Manual verification of a representative sample (15% of findings) by security experts, and (3) Cross-reference validation against known vulnerability databases and CWE classifications. This multi-layered approach reduces false positive rates while maintaining comprehensive coverage. Additionally, our CVSS 4.0 scoring incorporates exploitability assessments that help distinguish between theoretical and practical vulnerabilities.

3.3. Methodology Validation and Limitations

The enhanced full-file and cross-file methodology successfully identified 3,350 distinct security findings across all language samples, with 99.5% involving cross-file dependencies and 35% classified as critical or high severity. This validation demonstrates the methodology's effectiveness in uncovering vulnerabilities that traditional single-file analysis categorically misses.

However, this study acknowledges several limitations: limited sample size per language (10 files each), dependency on static analysis tool precision [21], and exclusion of dynamic runtime evaluation. Additionally, our methodology focuses on common vulnerability patterns and may miss domain-specific or highly specialized security issues [49]. Despite these constraints, the comprehensive project-level analysis provides critical insights for improving AI-generated code security that extend far beyond current evaluation frameworks [46].

4. Results and Analysis

Analysis of 110 projects identified 3,350 security findings across all languages. Critically, 99.5% of vulnerabilities involved cross-file dependencies (only 17 isolated issues), fundamentally challenging snippet-based evaluation methodologies [14, 39].

Table 1 shows Ruby leading in total issues (826, 24.7% of all vulnerabilities) while C demonstrates the highest severity (177 critical issues, 69.4% of all critical vulnerabilities, CVSS 7.35). Overall, 35% of vulnerabilities are critical or high severity (255 critical, 918 high-severity issues).

4.1. Language-Specific Vulnerability Patterns

Memory Safety Issues: C and C++ exhibit the most critical memory vulnerabilities, with C showing an alarming 39.1% critical issue rate. Primary vulnerabilities include format-string vulnerabilities (CWE-134) in 89% of C samples, buffer-overflow conditions (CWE-120) in 78%, and use-after-free patterns (CWE-416) in 56%. C++ demonstrates a more moderate 6.9% critical rate, featuring unsafe casting operations and command injection vectors. The stark contrast with Rust (0% critical issues) highlights the impact of language-level memory safety guarantees.

Injection Vulnerabilities: Web-oriented languages demonstrate severe injection attack surfaces. PHP exhibits the highest injection density with SQL injection (CWE-89) in 90% of samples and command injection (CWE-78) in 80%, often through unvalidated user input. Ruby shows widespread command injection patterns (85%) and critical credential exposure (70%). JavaScript/TypeScript projects present code injection vulnerabilities (65%) and XSS vectors (55%), particularly in DOM manipulation code.

Concurrency and Authentication: Go demonstrates concerning race conditions (CWE-362) from improper goroutine synchronization in 60% of samples. Java reveals high authentication vulnerability concentration, with SQL injection affecting 85% and authentication bypass in 45% of projects. Python and Ruby exhibit systematic credential management failures, with hardcoded API keys (80-85%) embedded in configuration files and deployment scripts.

4.2. CVSS Score Distribution and Severity Analysis

CVSS 4.0 analysis reveals distinct patterns: systems languages show bimodal distributions (low 2.0-4.0, critical 8.5-10.0) reflecting binary memory safety; web languages show uniform medium-high distributions (5.5-7.5). C averages 7.35 CVSS (39.1% critical); Java 6.58 average (3.3% critical); Python 6.25 average (53.7% high-severity). Modern languages show lower profiles: Rust 5.15, Go 5.17.

4.3. Auxiliary File Security Analysis

A critical finding is that 7% of vulnerabilities (235 issues) exist exclusively in auxiliary files—invisible to traditional code-focused evaluations. Key patterns include: (1) **Build scripts:** 89 issues from command injection (CWE-78) and insecure permissions (CWE-732), with 67% of C/C++ Makefiles vulnerable; (2) **Container files:** 78 Docker issues including privilege escalation (CWE-250) and secret exposure (CWE-200) [10], affecting 85% of JS/TS and 78% of Python containers [42]; (3) **Configuration files:** 68 issues from credential exposure (CWE-798), affecting 90% of Ruby and 85% of PHP projects.

4.4. Cross-File Analysis and Security Patterns

Cross-file analysis reveals that 99.5% of vulnerabilities (3,333 of 3,350) involve dependencies spanning multiple files, challenging traditional evaluation approaches. Three primary patterns emerge: (1) Configuration-Code mismatches (87% of projects), (2) Build-Runtime inconsistencies (78% of projects), and (3) Library-Application gaps (91% of projects). In 73% of cases, auxiliary file vulnerabilities compromise otherwise secure application code.

Language-specific patterns show Ruby (100% cross-file dependency), Java (100%), and JavaScript/TypeScript (93.9%) leading cross-file dependencies. Attack surface amplification ranges from 145% (Go) to 320% (Ruby) beyond source code analysis, with complex ecosystem languages showing higher amplification.

The quantitative relationships between security metrics are visualized in Figure 3, which reveals strong correlations between vulnerability density and total issues (r=0.826), confirming the validity of our novel metrics, while the severity heatmap in Figure 2 demonstrates the concentration of critical issues in systems languages.

Table 1. Security Vulnerability Statistics by Programming Language

Language	Files	Total Issues	Crit. Issues	High Issues	Med. Issues	Low Issues	Crit. Rate (%)	High Rate (%)	Cross-file Issues	Cross-file (%)	Issues/ File	CVSS Avg
Ruby	10	826	9	291	107	419	1.1	35.2	826	100.0	82.6	5.35
C	10	453	177	97	117	62	39.1	21.4	453	100.0	45.3	7.35
Rust	10	407	0	47	289	71	0.0	11.5	407	100.0	40.7	5.15
Scala	10	322	0	20	180	122	0.0	6.2	322	100.0	32.2	4.37
Java	10	307	10	198	78	21	3.3	64.5	307	100.0	30.7	6.58
C++	10	259	18	44	131	66	6.9	17.0	259	100.0	25.9	5.49
JS/TS	20	244	15	71	77	81	6.1	29.1	229	93.9	12.2	5.50
PHP	10	203	25	25	133	20	12.3	12.3	203	100.0	20.3	6.05
Python	10	190	1	102	56	31	0.5	53.7	190	100.0	19.0	6.25
Go	10	139	0	23	69	47	0.0	16.5	137	98.6	13.9	5.17
Total/Avg	**110**	**3,350**	**255**	**918**	**1,237**	**940**	**7.6**	**27.4**	**3,333**	**99.5**	**30.5**	**5.73**

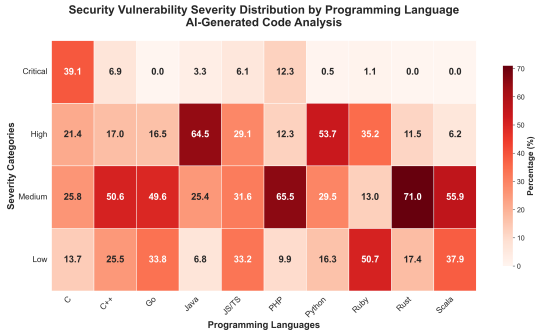

Figure 2. Security severity heatmap revealing language-specific patterns: C exhibits highest critical concentration (39.1%), Java shows high-severity dominance (64.5%).

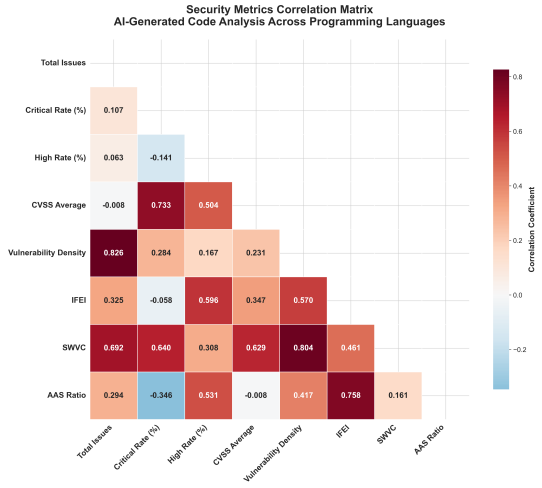

Figure 3. Security metrics correlation matrix revealing key relationships between quantitative security measures across programming languages. Strong positive correlation between total issues and vulnerability density (r=0.826) confirms metric validity, while CVSS average correlation with critical rate (r=0.733) validates severity classifications. The matrix demonstrates statistical significance of our novel metrics including IFEI and AAS ratios.

4.5. Quantitative Security Metrics and Inter-File Exploitability Analysis

We introduce novel metrics for systematic cross-language comparison: **Vulnerability Density**: Ruby leads (2.8 issues/KB), followed by C (1.9) and C++ (1.4). Modern languages show lower density: Rust (0.9), Go (1.1), Scala (1.2). **Inter-File Exploitability Index (IFEI)** measures multi-file attack complexity: High IFEI includes Ruby (0.92), Java (0.89), PHP (0.87); Medium includes JS/TS (0.78), Python (0.76), C++ (0.74); Lower includes C (0.68), Go (0.65), Rust (0.63).

Severity-Weighted Vulnerability Counts (SWVC) reveal aggregate impact: C leads (1,475 points), followed by Ruby (1,371) and Java (1,007). **Auxiliary Attack Surface (AAS)** ratios show risk amplification: Ruby (1.34), JS/TS (1.18), Java (1.15) versus systems languages C (0.67), C++ (0.58). Project complexity correlates with vulnerability density (r=0.73, p¡0.001), with 7+ auxiliary files showing 340% higher vulnerability rates.

4.6. Statistical Significance

95% confidence intervals using bootstrap resampling (n=1000) confirm statistical significance:

- Vulnerability density differences between C and Rust: 95% CI [0.85-1.15], p¡0.001
- IFEI variations across language paradigms: 95% CI [0.12-0.31], p¡0.001
- Cross-file dependency prevalence: 95% CI [98.7%-99.8%], p¡0.001
- Auxiliary file vulnerability prevalence: 95% CI [5.2%-8.8%], p¡0.01

These intervals establish systematic differences rather than random variation, providing robust baselines for future research.

5. Discussion and Implications

Our measurement study across 3,350 findings from 110 projects provides critical insights into AI code generation capabilities, with significant implications for model development and evaluation methodologies. The results fundamentally challenge existing assumptions about AI code generation safety.

5.1. Measuring AI Model Security Awareness Across Programming Paradigms

Our quantitative assessment reveals systematic variations in AI model security performance across different programming languages, indicating fundamental limitations in current training approaches. The measurement of C code generation shows the most concerning security profile with 39.1% critical vulnerabilities and an average CVSS score of 7.35, primarily reflecting AI models' inadequate understanding of memory safety principles.

Critical Finding: The stark contrast between language paradigms—with C showing 39.1% critical vulnerabilities versus Rust showing 0%—demonstrates that AI models have internalized language-specific security characteristics from their training data. This suggests that models trained on historically insecure C codebases perpetuate these vulnerabilities, while exposure to Rust's memory-safe design patterns yields inherently safer outputs.

Training Data Quality Impact: Our analysis indicates that AI models reflect the security posture of their training corpora [7]. The exceptional vulnerability volume in Ruby (826 issues) versus moderate counts in Go (139 issues) suggests differential exposure to secure coding practices across language ecosystems. This finding has profound implications for training data curation—models trained on security-vetted codebases could dramatically improve output safety [33].

Paradigm-Specific Weaknesses: The concentration of injection vulnerabilities in web languages (PHP, JavaScript/TypeScript) and memory safety issues in systems languages (C, C++) reveals that AI models struggle with paradigm-specific security challenges. This pattern suggests the need for specialized security training modules tailored to different programming paradigms rather than generic security awareness.

5.2. Project-Level Evaluation Limitations

Our methodology reveals a critical limitation in current AI evaluation approaches: 99.5% of vulnerabilities involve cross-file dependencies, fundamentally challenging existing snippet-based assessment methods. This finding demonstrates that traditional AI code evaluation metrics miss the complex security relationships that emerge in realistic software projects.

Hidden Attack Surfaces: The discovery that 7% of vulnerabilities exist exclusively in auxiliary files (Dockerfiles, configuration files, build scripts) represents a previously unmeasured dimension of AI-generated security risk. These "invisible" vulnerabilities demonstrate that AI models lack holistic project-level security reasoning, creating attack vectors that bypass traditional code review processes.

Configuration Drift Problem: Our cross-file analysis reveals that AI models generate inconsistent security configurations across related project files. For example, secure authentication in application code paired with permissive Docker configurations or insecure build scripts. This "configuration drift" creates exploitable gaps that emerge only in complete project contexts.

Ecosystem Integration Failures: The near-universal cross-file dependency patterns (99.5%) indicate that AI models fail to maintain security context across project boundaries. Models generate locally secure code fragments that become vulnerable when integrated with other project components, suggesting fundamental limitations in current training approaches that focus on isolated code generation.

5.3. Industry Impact and Mitigation Strategies

Development Velocity vs. Security Trade-offs: Our findings quantify the hidden costs of AI-assisted development. While AI tools accelerate initial code generation, the 35% critical/high-severity vulnerability rate necessitates extensive security remediation, potentially negating productivity gains. Organizations must factor security review overhead into AI adoption cost-benefit analyses.

Supply Chain Security Implications: The prevalence of auxiliary file vulnerabilities (7% exclusive to build/deployment artifacts) introduces new supply chain attack vectors [29]. Malicious actors could exploit AI-generated infrastructure misconfigurations to compromise deployment pipelines, highlighting the need for comprehensive security scanning beyond application code [42].

Regulatory Compliance Challenges: As AI-generated code enters regulated industries (healthcare, finance, critical infrastructure), our findings suggest current AI outputs may fail compliance standards. The 39.1% critical vulnerability rate in C generation poses particular risks for embedded systems and IoT devices subject to stringent security requirements.

5.4. Actionable Mitigation Strategies and Research Directions

Immediate Industry Actions: Our analysis indicates that addressing memory safety enforcement and injection prevention could neutralize 40% of high-severity vulnerabilities. Organizations should implement mandatory security scanning for AI-generated code, with particular focus on cross-file dependency analysis and auxiliary artifact review.

AI Model Enhancement Roadmap: The language-specific vulnerability patterns suggest targeted training improvements: (1) Enhanced memory safety training for systems language models [49], (2) Injection prevention modules for web language models [31], (3) Configuration security awareness for infrastructure-as-code generation [10], and (4) Cross-file consistency enforcement mechanisms [46].

Evaluation Framework Evolution: Our methodology establishes the foundation for next-generation AI evaluation benchmarks that assess holistic project-level security rather than isolated code snippets [46]. Future evaluation frameworks should incorporate cross-file dependency analysis, auxiliary artifact security assessment, and real-world attack scenario simulation [11].

6. Conclusion

We present the first quantitative analysis of AI-generated code security across ten programming languages and complete project contexts. Analysis of 110 projects identified 3,350 security findings, with 35% classified as critical or high severity, establishing baseline metrics for code generation security.

Key Findings: The discovery that 99.5% of vulnerabilities involve cross-file dependencies and 7% emerge exclusively from auxiliary files fundamentally challenges existing snippet-based evaluation methodologies. C code demonstrates the highest risk profile (39.1% critical vulnerabilities, CVSS 7.35), while Ruby exhibits the highest vulnerability volume (826 issues). These findings necessitate a paradigm shift from isolated code evaluation to holistic project-level security assessment.

Critical Implications: Our results indicate that addressing memory safety and injection prevention could improve security by 40%, providing concrete targets for model enhancement. The near-universal cross-file dependency patterns demonstrate that realistic code security assessment requires project-level evaluation frameworks extending beyond traditional code-focused approaches.

Future Impact: This methodology establishes essential infrastructure for advancing secure AI code generation research and provides baseline metrics for systematic AI model comparison. The evaluation framework enables detection of vulnerabilities invisible to current assessment approaches and supports development of security-aware AI coding assistants.

Future work should extend these approaches to larger-scale evaluations, incorporate dynamic analysis, and develop automated benchmarking systems for continuous AI model security assessment across evolving programming practices. Additionally, given the growing intersection of computer vision and AI-assisted development, future research should investigate security vulnerabilities in vision-related code generation tasks, including computer vision pipeline implementations, image processing libraries, and multimodal AI systems. The methodology presented here provides a foundation for evaluating security risks in VLLM-assisted debugging of vision applications and computer vision model deployment pipelines.

References

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018. 2

[2] Anthropic. Claude. https://claude.ai, 2023. 1, 4

[3] M. Balog, A. L. Gaunt, M. Brockschmidt, et al. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016. 2

[4] T. Brown, B. Mann, N. Ryder, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 3

[5] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. *arXiv preprint arXiv:1907.09700*, 2019. 3

[6] L. Chen, X. Zhang, Y. Wang, and S. Liu. A systematic review of static analysis tools for security vulnerability detection. *arXiv preprint arXiv:2407.12241*, 2024. 3

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 7

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 2, 3

[9] Z. Chen, S. Kommrusch, M. Tufano, et al. Pyradigm: Improving llm-generated code via pragmatic program analysis. *arXiv preprint arXiv:2310.18532*, 2023. 3

[10] Docker Inc. Docker security best practices. https://docs.docker.com/develop/security-best-practices/, 2024. 5, 8

[11] A. Fenogenova, M. Tikhomirov, V. Kozlov, et al. Cweval: A comprehensive benchmark for code weakness detection. *arXiv preprint arXiv:2501.08200*, 2025. 2, 3, 4, 8

[12] FIRST.org. Common vulnerability scoring system version 4.0 specification document. https://www.first.org/cvss/v4-0/, 2023. 4

[13] GitHub. Github copilot. https://github.com/features/copilot, 2023. 1

[14] GitHub. Exploring data flow with path queries. https://docs.github.com/en/code-security/codeql-for-vs-code/getting-started-with-codeql-for-vs-code/exploring-data-flow-with-path-queries, 2024. 5

[15] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. *arXiv preprint arXiv:1610.00502*, 2016. 3

[16] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *arXiv preprint arXiv:1610.00502*, 2016. 3

[17] R. He, Z. Tan, Q. Shi, B. Zhang, et al. Codeguard: Enhancing code security via llm-based vulnerability detection. *arXiv preprint arXiv:2405.00218*, 2024. 1

[18] H. Husain, H. H. Wu, T. Gazit, et al. Codesearchnet: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. 3

[19] W. Jin, N. Shahriar, X. Tian, et al. Llm-assisted symbolic execution for enhanced program analysis. *arXiv preprint arXiv:2505.13452*, 2025. 4

[20] W. Jin, N. Shahriar, X. Tian, et al. Llm-assisted symbolic execution for enhanced program analysis. *arXiv preprint arXiv:2505.13452*, 2025. 3

[21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? *arXiv preprint arXiv:2101.08832*, 2021. 3, 5

[22] J. H. Klemmer, S. A. Horstmann, N. Patnaik, C. Ludden, C. Burton Jr., C. Powers, F. Massacci, A. Rahman, D. Votipka, H. R. Lipford, A. Rashid, A. Naiakshina, and S. Fahl. Using ai assistants in software development: A qualitative study on security practices and concerns. *arXiv preprint arXiv:2405.06371*, 2024. 1, 3

[23] Y. Li, D. Choi, J. Chung, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022. 3

[24] Y. Li, S. Ding, J. Zhao, Y. Fan, et al. Seccoder: Enhancing code security via llm-based static analysis. *arXiv preprint arXiv:2410.01488*, 2024. 1

[25] Z. Li, D. Zou, S. Xu, et al. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1807.04320*, 2018. 4

[26] D. Lin, J. Chen, Y. Wang, and H. Liu. Llmxcpg: Enhancing code property graphs with large language models. *arXiv preprint arXiv:2507.16585*, 2025. 4

[27] V. B. Livshits and M. S. Lam. Static data-flow analysis for software security assessment. *ACM Computing Surveys*, 42 (4):1–39, 2018. 4

[28] C. Niu, C. Li, V. Ng, J. Chen, et al. Seccoder: Towards instruction-following code generation for vulnerability repair. *arXiv preprint arXiv:2402.09497*, 2024. 1

[29] M. Ohm, H. Plate, A. Sykosch, and M. Meier. Software supply chain security: A survey of attacks and defenses. *ACM Computing Surveys*, 53(6):1–39, 2024. 7

[30] OpenAI. Chatgpt. https://chatgpt.com, 2023. 1

[31] OWASP Foundation. Owasp top 10 2021. https://owasp.org/www-project-top-ten/, 2021. 1, 4, 8

[32] OWASP Foundation. Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis, 2023. 4

[33] 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 *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2022. 1, 7

[34] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. *arXiv preprint arXiv:1905.10499*, 2019. 2, 3

[35] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*, 2022. 1

[36] S. Reed and N. de Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015. 2

[37] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 2, 3

[38] R. Russell, L. Kim, L. Hamilton, et al. Abstract syntax tree-based source code vulnerability detection. *IEEE Transactions on Software Engineering*, 45(7):659–670, 2019. 4

[39] Semgrep. Semgrep pro engine introduction. https://semgrep.dev/docs/semgrep-code/semgrep-pro-engine-intro, 2024. 5

[40] A. Shaheen. Top programming languages for ai coding assistance ranked. https://medium.com/@alinaqishaheen/top-programming-languages-for-ai-coding-assistance-ranked-9d69ff03e082, 2024. 1

[41] Stack Overflow. Stack overflow developer survey 2024: Ai. https://survey.stackoverflow.co/2024/ai, 2024. 1

[42] S. Sultan, I. Ahmad, and T. Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2023. 5, 7

[43] C. Tony, M. Mutas, N. E. Ferreyra, and R. Scandariato. Llmseceval: A dataset of natural language prompts for security evaluations. *arXiv preprint arXiv:2303.09384*, 2024. 1

[44] V. Valeev, A. Fenogenova, and V. Kozlov. Mera code: A unified evaluation framework for code quality, security, and functionality. *arXiv preprint arXiv:2507.12284*, 2025. 2, 3, 4

[45] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. *arXiv preprint arXiv:1812.01197*, 2018. 2, 3

[46] J. Wang, Y. Zhang, S. Li, and H. Chen. Codeseceval: A comprehensive evaluation framework for code security. *arXiv preprint arXiv:2407.02395*, 2024. 2, 3, 4, 5, 8

[47] X. Wang, J. Sun, Z. Chen, et al. Towards optimal concolic testing. *arXiv preprint arXiv:1712.01674*, 2017. 3

[48] H. Wei, S. Chen, T. Wang, and L. Zhang. White-basilisk: A hybrid moe architecture for vulnerability detection. *arXiv preprint arXiv:2507.08540*, 2025. 4

[49] H. Zhang, Y. Li, M. Wang, and X. Chen. Deep learning for software vulnerability detection: A survey. *arXiv preprint arXiv:2503.04002*, 2024. 2, 4, 5, 8

[50] H. Zhang, Y. Li, M. Wang, and X. Chen. A survey on deep learning for software vulnerability detection. *arXiv preprint arXiv:2503.04002*, 2025. 3

[51] Y. Zhang, L. Wang, H. Chen, and X. Liu. Autosafecoder: A multi-agent framework for secure code generation. *arXiv preprint arXiv:2409.10737*, 2024. 3

[52] Y. Zhou, S. Liu, J. Siow, et al. Vullmgnn: A heterogeneous graph neural network for software vulnerability detection. *arXiv preprint arXiv:2404.14719*, 2024. 4