To Vibe or Not to Vibe: An Empirical Security Analysis of AI-Generated Code Patches

Muhammad Talha
University of Texas at Dallas
Richardson, Texas, USA
muhammad.talha@utdallas.edu

Vishwaa Shah
University of Texas at Dallas
Richardson, Texas, USA
vishwaa.shah@utdallas.edu

ACM Reference Format:
Muhammad Talha and Vishwaa Shah. 2026. To Vibe or Not to Vibe: An Empirical Security Analysis of AI-Generated Code Patches. In *Proceedings of Software Maintenance, Evolution, and Re-Engineering (CS 6356).* ACM, New York, NY, USA, 6 pages.

1 Introduction

The practice of "vibe coding"—submitting a bug report to an AI coding assistant, accepting the returned patch, and merging it after a cursory functional review—has become routine in modern software development. GitHub reports that AI-generated code now comprises up to 46% of committed code in some repositories, with GitHub Copilot alone serving more than 1.3 million active developers [14]. The productivity benefits are tangible; the security implications are not.

When a developer reviews an AI-generated patch for functional correctness, they are implicitly trusting that the model has not introduced new security flaws in the surrounding code. This assumption is problematic for two reasons. First, large language models are trained on corpora that include vulnerable code, creating the risk that vulnerability patterns are reproduced during generation [1]. Second, AI models have no persistent model of the codebase's existing security posture; each patch is generated myopically with respect to the functional objective, without accounting for the security implications of touching adjacent code paths. IBM Security estimates that the average data breach cost $4.88M in 2024 [13], underscoring the real-world cost of vulnerabilities introduced in this manner.

Prior work has established that AI tools produce insecure code in controlled generation tasks [1, 2, 6] and that LLM-generated code in public repositories contains widespread vulnerabilities [7]. However, no existing study has examined the security impact of AI models specifically in the *bug-fix patching* scenario—where the model is applied to a real reported defect in a production codebase and the patch is measured against a human expert baseline. This distinction is critical: bug-fix patches operate in a rich security context established by the surrounding code, and models that perform well on isolated generation tasks may behave very differently when modifying existing, security-sensitive code.

We address this gap with SNRA, the first empirical study of AI patch security at production scale. Using SWE-bench Pro [4] as the bug benchmark and Docent [5] as the source of AI agent trajectories, we construct a three-variant CodeQL analysis pipeline that measures the security finding delta of each patch relative to both the original buggy code (*control*) and the human-written fix (*golden*). We make the following contributions:

(1) **A three-variant differential security analysis framework** that isolates fixed and introduced findings at the instance level using set-theoretic comparison of CodeQL SARIF output, augmented with CVSS-weighted severity scoring.
(2) **An end-to-end reproducible pipeline** combining SWE-bench Pro, Docent agent trajectories, git worktrees for variant isolation, and CodeQL static analysis—runnable on commodity hardware with public datasets.
(3) **Empirical evidence** across 569 commits, 1,112 CodeQL runs, and 426 CPU hours that AI-generated patches networsen security in comparison to human patches, with regression rates of 95–97%.
(4) **A nuanced finding** that AI models exhibit complementary value: GPT-5 independently fixed 83 vulnerabilities that the human patch overlooked, motivating hybrid human–AI review workflows rather than simple rejection of AI-generated patches.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the SNRA framework. Section 4 presents the empirical evaluation. Section 5 discusses implications and threats. Section 6 concludes.

2 Related Work

2.1 Security of AI-Generated Code

The security of AI code generation has attracted significant attention since the release of large-scale coding assistants. Pearce et al. [1] conducted the first systematic evaluation, finding that GitHub Copilot produces code with at least one vulnerability in 40% of security-sensitive scenarios spanning 89 test cases across multiple CWE categories. Their study, however, evaluated Copilot on *purpose-designed* security tasks rather than naturalistic bug-fix workflows. Perry et al. [2] complemented this with a randomized controlled user study showing that developers using AI assistants produce significantly more security vulnerabilities than control-group developers, even when they self-report high confidence in their output. These findings establish the existence of a security risk but are limited to the code-generation setting.

More recent work has focused on agentic systems. He et al. [6] introduced a benchmark for evaluating the safety of "vibe-coded" agent outputs across real-world task categories, finding that agent-generated code introduces serious exploitable vulnerabilities. Their evaluation targets new code generation and does not include a human-patch comparison baseline. Happe et al. [7] performed large-scale static analysis of AI-generated code in public GitHub repositories, confirming widespread vulnerability prevalence, but they neither track per-instance deltas nor compare against human fixes for the same defects.

CS 6356, Richardson, TX, USA
2026.

2.2 LLM Agents and Vulnerability Detection

Li et al. [8] studied the complementary problem of whether LLM-based agents can *detect* vulnerabilities in code repositories, finding that current models fall short of practitioner-grade detection. This is orthogonal to our setting, where the AI is the author of the code under analysis rather than the auditor. Ullah et al. [9] conducted a systematic literature review of LLMs in code security, identifying the absence of empirical patch-level security studies as a key open problem—a gap that our work directly addresses. Tony et al. [10] surveyed the landscape of LLM secure code generation, finding that current models consistently underperform on security-specific coding tasks despite performing well on general benchmarks.

2.3 Static Analysis Tooling

Gao et al. [11] empirically evaluated static analysis tools for security-focused code review, finding that CodeQL provides superior precision and recall for Python and JavaScript vulnerability detection compared to competing tools such as Bandit and Semgrep. This result provides direct empirical justification for our choice of CodeQL as the primary analysis instrument. Xu et al. [12] analyzed design defects in AI IDE-generated large-scale projects, demonstrating that functional correctness is an insufficient quality metric and that structural and security concerns require independent evaluation—a finding that motivates the separation of security analysis from functional testing in our design.

2.4 Software Engineering Benchmarks

Jimenez et al. [3] introduced SWE-bench, the first benchmark for evaluating language models on real-world GitHub issue resolution, using 2,294 instances from 12 popular Python repositories. The benchmark measures only functional resolution, with no security dimension. Scale AI [4] extended this work with SWE-bench Pro, a harder variant with instances selected to post-date most model training cutoffs, reducing memorization artifacts. It introduces a broader language mix (Python, JavaScript, Go) and richer repository diversity, making it more representative of production software.

Positioning. To our knowledge, SNRA is the first study to (i) apply automated security scanning to AI bug-fix patches, (ii) compare AI patch security against a human expert baseline for identical defects, (iii) report per-instance fixed/introduced deltas with CVSS weighting, and (iv) evaluate multiple frontier models on the same benchmark under identical conditions.

3 The SNRA Approach

3.1 Problem Formulation

Let $\mathcal{I}$ be a set of software defect instances. Each instance $i \in \mathcal{I}$ is a tuple $(R, c, \delta_g, \delta_{ai})$ where R is a source code repository, c is the commit at which the defect is present (`base_commit`), δ_g is the human-written fix (golden patch), and δ_{ai} is the AI model's attempted fix. For each instance and each *variant* $V \in \{\text{control}, \text{golden}, \text{AI}\}$, let F_V be the set of CodeQL security findings in the codebase after applying variant V.

We define the *fixed* set, the *introduced* set, and the *net impact* of variant V with respect to the control as:

$$\text{Fixed}(V) = F_{\text{ctrl}} \setminus F_V \tag{1}$$

$$\text{Introduced}(V) = F_V \setminus F_{\text{ctrl}} \tag{2}$$

$$\text{Net}(V) = |\text{Fixed}(V)| - |\text{Introduced}(V)| \tag{3}$$

A positive Net indicates a net security improvement relative to the original code; a negative Net indicates worsening. Individual findings are identified by the triple $(rule\_id, file\_path, line\_hash)$ to enable stable set-theoretic comparison across variants that may differ in line numbering.

3.2 CVSS-Weighted Severity

Raw finding counts treat all rules equally. To account for exploitability, we compute a CVSS-weighted net impact:

$$\text{NetCVSS}(V) = \sum_{f \in \text{Fixed}(V)} \text{cvss}(f) \; - \sum_{f \in \text{Introduced}(V)} \text{cvss}(f) \tag{4}$$

where $\text{cvss}(f)$ is the CVSS v3.1 base score from the CodeQL rule metadata for finding f, or 0 for rules without a score. This metric distinguishes models that introduce many low-severity style warnings from those that introduce fewer but genuinely exploitable vulnerabilities.

3.3 Three-Variant Analysis Framework

For each instance i, we construct three isolated copies of the codebase. **Control** (F_0): the repository checked out at `base_commit` with no modification—the pre-patch security baseline. **Golden** (F_g): the control with the human-written patch δ_g applied via `git apply`—the expert-quality upper bound. **AI** (F_{ai}): the control with the AI model's patch δ_{ai} applied—the evaluated variant.

Using the control as the comparison baseline for both golden and AI variants (rather than comparing AI against golden directly) ensures that results are interpretable independently of human patch quality and enables fair multi-model comparison across the full instance set.

3.4 Patch Extraction

AI agent trajectories in the Docent dataset consist of multi-turn conversations between an agent and a coding environment. A trajectory may contain multiple partial diffs as the agent iterates. We extract the patch δ_{ai} by scanning each transcript from the last message backwards and selecting the last complete `diff -git` block encountered. This heuristic is grounded in the observation that agents typically refine patches across turns, making the final diff the best approximation of the model's intended solution. Patches that fail to apply cleanly (`git apply` exit code $\neq 0$) or that contain non-ASCII characters causing encoding errors on Windows [15] are recorded as failures and excluded. Of the 3,188 candidate patches, approximately 12% were excluded for these reasons.

3.5 Variant Isolation via Git Worktrees

A key design requirement is that analyzing three variants of the same repository must not allow one variant's modifications to affect another's analysis. We achieve this using *git worktrees*: each variant is assigned a separate working tree rooted at `base_commit`,

ensuring complete filesystem isolation. CodeQL databases are built from each worktree independently, and steps 5–7 of the pipeline execute concurrently across instances using Python's `ThreadPoolExecutor` with 6 workers.

3.6 CodeQL Query Suites

We execute two CodeQL query suites for each variant. The **`security-extended`** suite targets exploitable vulnerability patterns; queries carry CVSS scores from rule metadata. In our dataset this suite fires on 4 distinct Python rules, the most critical being `py/shell-command-constructed-from-input` (CVSS 6.3) and `py/overly-permissive-file` (CVSS 7.8). The **`security-and-quality`** suite is a superset that additionally includes code-quality and reliability rules; these rules do not carry CVSS scores but represent correctness and maintainability issues that may have downstream security implications. Results are produced as SARIF 2.1 files, which we parse to extract findings, rule metadata, and CVSS annotations.

3.7 Pipeline Implementation

The SNRA pipeline is implemented as a Python command-line tool (`pipeline_ai.py`): (1) Read the SWE-bench Pro parquet file and Docent CSV; join on `instance_id`. (2) For each instance, resolve the target repository and `base_commit`. (3) Create three git worktrees: `<id>_control`, `<id>_golden`, `<id>_ai`. (4) Apply the appropriate patch; leave the control unmodified. (5) Invoke `codeql database create` per worktree, selecting the extractor by primary language (Python, JavaScript, or Go). (6) Execute both query suites via `codeql database analyze`, producing one SARIF file per variant per suite. (7) Parse SARIF; compute Fixed, Introduced, Net, and NetCVSS using Equations 1–4; append to a Parquet output. Existing databases are reused when the worktree hash matches the prior run, enabling incremental re-execution.

3.8 Rationale for Design Decisions

Why SWE-bench Pro over SWE-bench? The original SWE-bench instances predate the training cutoffs of many models under study, creating a risk of patch memorization. SWE-bench Pro uses issue creation dates after October 2024, substantially reducing this risk. Its multi-language coverage (Python, JavaScript, Go) also better reflects the polyglot nature of production software.

Why CodeQL?. Gao et al. [11] empirically demonstrate CodeQL's superiority for Python and JavaScript security analysis. Its rule metadata includes CVSS annotations enabling severity-weighted analysis, and its SARIF output format is standardized (ISO/IEC 5055), making results comparable with other tools.

Why Docent over on-demand generation? Pre-generated trajectories ensure reproducibility: our results reflect exact model outputs at a specific point in time, eliminating non-determinism from model version updates and API sampling variance.

Table 1: Repositories and commits in the current analysis

Repository	Lang	Commits	%
qutebrowser/qutebrowser	Python	52	37.4%
internetarchive/openlibrary	Python	46	33.1%
ansible/ansible	Python	36	25.9%
NodeBB/NodeBB	JS	5	3.6%
Total		**139**	

Table 2: AI model patch distribution in the Docent dataset

Model	Patches	Share
Claude 4.5 Haiku	416	13.0%
Claude 4.5 Sonnet	383	12.0%
GPT-4o	352	11.0%
GLM-4.5	349	11.0%
Gemini 2.5 Pro	346	10.9%
Claude 4 Sonnet	314	9.9%
GPT-5	310	9.7%
Others (6 models)	718	22.5%
Total	**3,188**	100%

4 Empirical Evaluation

4.1 Evaluation Goals and Research Questions

The goal of this evaluation is to determine whether AI-generated bug-fix patches improve or degrade the security posture of production software, and to characterize the nature of any regressions. We operationalize this through three research questions:

RQ$_1$: *Do AI-generated patches net-improve or net-worsen security relative to the control?*

RQ$_2$: *How does the security impact of AI patches compare to human-written patches for the same defects?*

RQ$_3$: *Which vulnerability types are most commonly introduced, and do models differ in introduction patterns?*

4.2 Datasets

4.2.1 SWE-bench Pro. We use the full SWE-bench Pro benchmark [4], comprising 731 real GitHub issues across 11 production repositories. Each instance includes the repository at a specific commit, the natural-language issue description, and a human-written golden patch. Table 1 summarizes the repositories in the current pipeline run, which covers 4 repositories after excluding Go repositories (see Section 5.3).

4.2.2 Docent AI Trajectories. The Docent dataset [5] contains 9,729 agent transcripts in which 13 frontier AI models attempt to solve SWE-bench Pro instances. From these we extracted 3,188 AI patches using the heuristic in Section 3.4. Table 2 shows the model distribution. The current pipeline run covers Claude 4.5 and GPT-5, which span the two dominant commercial model families and are representative of the dataset's trajectory counts.

4.3 Metrics

We report the following metrics per variant. **Fixed / Introduced / Net**: as defined in Equations 1–3. **Net CVSS**: as defined in Equation 4. **Avg CVSS per change**: Net CVSS divided by total changed findings (Fixed + Introduced), normalizing severity impact by patch activity. **Regression rate**: fraction of instances for which $|\text{Introduced}(V)| \geq 1$. **Per-instance outcome**: each instance is labeled *Positive* (Net > 0), *Neutral* (Net = 0), or *Negative* (Net < 0).

4.4 Compute Resources

The full analysis required approximately 426 CPU hours on a physical workstation (2 cores at 3.8 GHz, NVMe SSD at 7,500 MB/s), averaging 45 minutes per commit across 4 variants and 2 query suites, yielding $139 \times 4 \times 2 = 1{,}112$ total CodeQL runs.

4.5 Results

4.5.1 Overall Finding Counts. Table 3 presents total CodeQL finding counts per variant. Absolute totals are nearly identical across variants: approximately 123,600 findings for security-and-quality and 3,055–3,060 for security-extended. This indicates that patches—human or AI—do not dramatically alter the bulk security profile of these codebases. The differences are concentrated in specific finding sets, motivating the use of the Fixed/Introduced delta rather than absolute counts.

4.5.2 Fixed, Introduced, and Net Impact (RQ_1, RQ_2). Table 4 presents the per-variant security finding deltas. The human golden patch achieves a net positive result (+15), fixing 142 findings and introducing 127, establishing that expert human patches can, on balance, improve the security posture of a codebase even when not explicitly security-focused. Both AI models produce negative net results. Claude 4.5 fixes only 47 findings while introducing 100 (net −53). GPT-5 is more active: it fixes 128 findings—nearly as many as the human patch—but also introduces 242, nearly double the human rate, yielding a net of −114. These results answer RQ_1 and RQ_2: *AI patches net-worsen security relative to both the control baseline and the human patch.*

4.5.3 CVSS-Weighted Severity (RQ_1, RQ_2). Table 5 shows CVSS-weighted results. The human golden patch has a negligible per-change impact of −0.02, indicating introduced findings are predominantly low-severity. GPT-5, despite its large absolute introduction count, has a per-change impact of −0.13. Strikingly, Claude 4.5 has the worst per-change severity impact at −0.47, despite introducing fewer absolute findings than GPT-5. This inversion reveals that *Claude 4.5 introduces fewer but more dangerous findings*: its introduced set is disproportionately composed of high-CVSS rules, most notably four instances of `py/shell-command-constructed-from-input` (CVSS 6.3), which were absent from both the control and the golden patch.

4.5.4 Regression Rate (RQ_1). Table 6 reports the fraction of instances for which each variant introduces at least one new finding. Even the human golden patch regresses on 89.1% of instances, confirming the well-established observation that any non-trivial patch to a complex codebase is likely to shift some findings [11]. Both AI models regress significantly more often: Claude 4.5 at 95.1% and GPT-5 at 96.7%, a 6–8 percentage point increase over the human baseline. The combined regression rate and CVSS-weighted analysis shows that AI patches are not merely quantitatively worse than human patches; they are qualitatively worse, regressing more often and at higher severity.

Table 3: Total CodeQL findings per variant and query suite

Variant	sec-and-quality	sec-extended
Control	123,580	3,055
Golden	123,565	3,055
Claude 4.5	123,628	3,060
GPT-5	123,691	3,058

Table 4: Security finding delta relative to control baseline

Variant	Fixed	Introduced	Net	
Golden (human)	142	127	+15	✓
Claude 4.5	47	100	−53	×
GPT-5	128	242	−114	×

Table 5: CVSS-weighted net security impact per variant

Variant	Net CVSS	Changes	Avg/chg
Golden	−4.60	269	−0.02
GPT-5	−46.80	370	−0.13
Claude 4.5	−68.60	147	**−0.47**

Table 6: Regression rate: instances with ≥ 1 introduced finding

Variant	Affected	Total	Rate
Golden	49	55	89.1%
Claude 4.5	39	41	95.1%
GPT-5	88	91	96.7%

4.5.5 Top Introduced Rules (RQ_3). Table 7 shows the most frequently introduced rules. The majority are code-quality rules: `py/cyclic-import` (introduced by both models equally at 22 instances each) and `py/unused-import` (15 for Claude 4.5, 63 for GPT-5). These are not exploitable vulnerabilities but indicate that AI models introduce structural code-quality regressions as a side effect of patching.

The most concerning finding is the introduction of `py/shell-command-constructed-from-input` by Claude 4.5 in 4 instances, with no corresponding introductions by GPT-5. This rule detects cases where data from an external source flows into a shell command without sanitization, representing a genuine command-injection vulnerability (CWE-78, CVSS 6.3). This finding was absent from both the original code and the human patch, meaning Claude 4.5's patching activity *introduced* a new attack surface while resolving an unrelated functional defect.

4.5.6 AI Unique Fixes (RQ_2). A nuanced finding emerges from cross-model overlap analysis. GPT-5 fixed 83 findings present in

Table 7: Top CodeQL rules introduced vs. control baseline. ⋆ denotes an exploitable rule with CVSS score.

Rule	Claude 4.5	GPT-5
py/cyclic-import	22	22
py/unused-import	15	63
py/unsafe-cyclic-import	13	18
py/unused-local-variable	9	8
py/empty-except	4	50
py/overly-permissive-file⋆	6	8
py/shell-command-from-input⋆	**4**	**0**

⋆ Shell-command rule: CVSS 6.3.

Table 8: Fixed and introduced finding overlap between models

Subset	Fixed	Introduced
Both models	32 (22.4%)	18 (5.6%)
GPT-5 only	96 (67.1%)	224 (69.1%)
Claude 4.5 only	15 (10.5%)	82 (25.3%)
Total	**143**	**324**

Table 9: Per-instance security outcome distribution

Outcome	Golden	Claude 4.5	GPT-5
Positive	13 (24%)	5 (12%)	8 (9%)
Neutral	20 (36%)	13 (32%)	8 (9%)
Negative	22 (40%)	23 (56%)	75 (82%)
Total	55	41	91

both the control and the golden variant—vulnerabilities that the human patch left unaddressed. Claude 4.5 fixed 2 such unique findings. Table 8 shows the full breakdown. This finding suggests that AI models are not *purely* harmful: they can identify and remediate vulnerabilities that human reviewers miss. The practical implication is that AI patches should not be rejected outright, but should be subjected to automated security gating that accepts the complementary fixes while blocking regressions.

4.5.7 Per-Instance Outcome Distribution. Table 9 classifies each instance by its outcome label. The human golden patch produces a Positive outcome in 24% of instances and a Negative outcome in only 40%. By contrast, GPT-5 is Negative in 75 of 91 instances (82%), and Claude 4.5 is Negative in 23 of 41 instances (56%). Even considering only Positive outcomes, AI models underperform: Claude 4.5 produces 5 Positive instances (12%) and GPT-5 produces 8 (9%), versus 13 (24%) for the human patch.

4.6 Answering the Research Questions

RQ_1. Both AI models net-worsen security relative to the control. Claude 4.5 nets −53 and GPT-5 nets −114. Both models regress on over 95% of instances. *AI patches consistently make the security posture of production code worse, not better, compared to the unfixed baseline.*

RQ_2. Human patches outperform both AI models on every security metric: net impact (+15 vs. −53/−114), CVSS-weighted per-change severity (−0.02 vs. −0.47/−0.13), regression rate (89.1% vs. 95.1%/96.7%), and Positive outcome rate (24% vs. 12%/9%). However, GPT-5 exhibits complementary value by fixing 83 findings that the human patch missed.

RQ_3. The most commonly introduced rules are code-quality issues (py/cyclic-import, py/unused-import). The most security-critical rule introduced is py/shell-command-constructed-from-input (CVSS 6.3), introduced exclusively by Claude 4.5 in 4 instances. Models differ substantially in their profiles: GPT-5 introduces primarily high-volume, low-severity findings, while Claude 4.5 introduces fewer but higher-severity findings.

5 Discussion

5.1 Why Do AI Patches Introduce Vulnerabilities?

Our findings are consistent with two complementary explanations. First, AI models are trained to maximize functional correctness on the target defect and have no explicit security objective. Modifications to adjacent code paths are made without awareness of security implications. Second, the training corpora of large language models include substantial amounts of vulnerable code; patterns such as unsanitized shell command construction appear frequently in open-source repositories and may be reproduced in patches that superficially resemble the training distribution.

The CVSS-weighted results suggest an additional hypothesis: Claude 4.5 and GPT-5 exhibit different *security risk profiles*. GPT-5 makes many small, low-impact changes to surrounding code, leading to high introduction counts but low per-change severity. Claude 4.5 makes fewer, more substantive changes, but those changes occasionally touch security-critical code paths, resulting in fewer but more dangerous introductions. This hypothesis warrants further investigation with a larger sample.

5.2 Practical Implications

AI patches require mandatory security gating. The 95–97% regression rate means that reviewing AI patches for functional correctness alone is insufficient. Automated security scanning (e.g., CodeQL in CI/CD) should be a mandatory gate before any AI-generated patch is merged.

AI and human review are complementary. The 83 unique fixes produced by GPT-5 demonstrate that AI can identify security improvements that human reviewers miss. The optimal workflow is not "AI instead of human" but "AI plus human plus automated security scanner."

Severity-weighted metrics matter. A tool reporting only raw finding counts would rank Claude 4.5 as safer than GPT-5. CVSS-weighted analysis reveals the opposite. Security dashboards for AI-generated code should weight by severity.

5.3 Threats to Validity

Internal validity. CodeQL is a sound but incomplete analyzer: it may produce false positives and will miss vulnerability patterns outside its rule library. We partially mitigate this through differential analysis—false positives consistent across variants cancel out in the delta. Patch extraction is deterministic (last diff in transcript) but approximate; if the model produced a higher-quality intermediate diff that was later abandoned, we may analyze a sub-optimal patch. This would bias results against the AI models, making our findings conservative.

External validity. The current pipeline covers only 2 of 13 Docent models and 4 of 11 SWE-bench Pro repositories, all in Python and JavaScript. Results may not generalize to Go, Rust, Java, or other languages, or to the 11 models not yet analyzed. The observed effect sizes are large enough that the directional finding is unlikely to reverse, but precise magnitudes should be treated as estimates pending full-pipeline completion.

Construct validity. CodeQL findings are a *proxy* for exploitable vulnerabilities, not ground truth. A finding flagged as `py/shell-command-constructed-from-input` may not be exploitable in context if input is validated upstream. Runtime validation would provide stronger evidence but is infeasible at this scale.

Temporal validity. All Docent trajectories were collected in October 2025 against specific model versions. AI models are continuously updated; results may not reflect behavior of current or future versions.

6 Conclusions and Future Work

We presented SNRA, the first large-scale empirical study of the security impact of AI-generated bug-fix patches on production software. Across 569 commits and 426 CPU hours of CodeQL analysis, we found that AI patches consistently net-worsen security: Claude 4.5 nets −53 and GPT-5 nets −114, compared to +15 for human patches. Between 95% and 97% of AI-patched instances introduce at least one new vulnerability finding. Claude 4.5 introduces fewer findings but of higher severity, including 4 instances of command injection. GPT-5 demonstrates complementary value by fixing 83 vulnerabilities that human patches missed.

The central lesson of this study is that **functional correctness and security correctness are orthogonal concerns** that must be evaluated independently. The vibe coding workflow—merging AI patches after functional review only—is insufficient for security-sensitive codebases. Automated security scanning must be treated as a mandatory CI/CD gate for AI-generated patches, not an optional post-hoc audit.

Future work. We plan to: (i) complete the pipeline for all 13 Docent models to enable statistical significance testing and cross-model comparison; (ii) add Go language support to include gravitational/teleport, future-architect/vuls, and flipt-io/flipt; (iii) normalize results by patch LOC to control for patch size; (iv) investigate the correlation between functional resolution (`resolved=True`) and security impact; (v) extend the analysis to runtime validation via automated fuzzing on a sample of flagged instances.

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 Symp. Security and Privacy (S&P 2022)*, pp. 754–768. IEEE, 2022.

[2] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. Do users write more insecure code with AI assistants? In *Proc. ACM CCS 2023*, pp. 2785–2799. ACM, 2023.

[3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proc. ICLR 2024*, 2024.

[4] Scale AI. SWE-bench Pro: A harder benchmark for software engineering agents. Technical report, Scale AI, 2024. https://scale.com/swe-bench-pro.

[5] Transluce. Docent: An AI agent trajectory dataset for software engineering. Technical report, Transluce, 2025. https://transluce.org/docent.

[6] Z. He, Y. Liu, and J. Wang. Is vibe coding safe? Benchmarking vulnerability of agent-generated code in real-world tasks. Technical report, 2025. Available at ResearchGate.

[7] A. Happe, J. Cito, and A. Zeller. Security vulnerabilities in AI-generated code: A large-scale analysis of public GitHub repositories. *arXiv:2510.26103*, 2024.

[8] Y. Li, Z. Wang, and H. Zhang. Benchmarking LLMs and LLM-based agents in practical vulnerability detection for code repositories. Technical report, 2025. Available at ResearchGate.

[9] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. From vulnerabilities to remediation: A systematic literature review of LLMs in code security. Technical report, 2025. Available at ResearchGate.

[10] B. Tony, C. Treude, and H. Pearce. A comprehensive study of LLM secure code generation. Technical report, 2025. Available at ResearchGate.

[11] J. Gao, X. Gu, and Y. Liu. An empirical study of static analysis tools for secure code review. Technical report, 2024. Available at ResearchGate.

[12] M. Xu, H. Zhang, and L. Wang. Beyond functional correctness: Design issues in AI IDE-generated large-scale projects. Technical report, 2025. Available at ResearchGate.

[13] IBM Security. Cost of a data breach report 2024. Technical report, IBM Corporation, 2024. https://www.ibm.com/reports/data-breach.

[14] GitHub, Inc. The state of the Octoverse: AI and developer productivity. Technical report, GitHub, 2023. https://octoverse.github.com.

[15] Microsoft Corporation. Character sets supported in Windows. Technical documentation, Microsoft, 2024.