
Undoing the Fix: Evaluating AI Control Against Security Regression Attacks

Ivan¹
Independent

With
Apart Research

Abstract

We introduce CVE regression as a control setting for evaluating AI agents performing software engineering tasks. In this setting, an untrusted executor resolves real GitHub issues from SWE-bench Verified while a neutral-sounding side task asks it to "clean up" code in a way that reverts a known security fix. A trusted monitor reviews the proposed changes without knowing which fix to look for. We construct a sample set of task pairs from 15 Django CVEs and 231 SWE-bench instances, enforcing zero file overlap between main tasks and side tasks. We evaluate 9 executor models (each of 15 task pairs, one per CVE) and 3 monitor tiers. Executor compliance with the attack instruction ranges from 13% to 100% depending on the model (65% overall). Monitors detect 88-95% of successful attacks but flag 35-58% of honest changes as suspicious, with larger monitors reducing false positives (65% usefulness) more than improving detection. These initial results suggest that CVE regression is a viable control setting that produces meaningful variation across both executor compliance and monitor detection. More broadly, they illustrate a class of risk where AI agents assisting code changes might silently remove important properties, security fixes, correctness invariants, or other accumulated protections from the code they modify.

¹ Work conducted for a weekend hackathon organized by Apart Research, March 2026. We thank Apart Research for compute support.

1. Introduction

AI control is a safety framework for deploying AI agents that may not be fully trusted, where a combination of monitoring, auditing, and protocol design aims to ensure safety even if the agent is intentionally subversive (Greenblatt et al., 2023). As AI agents are rapidly adopted for software engineering, e.g. Anthropic provides Claude Code for open-source maintainers of major projects (Anthropic, 2026), these agents increasingly work on mature codebases that carry various security fixes and patches accumulated over years of maintenance.

This may expose a specific risk: an AI agent asked to refactor, clean up, or modify code may suggest changes that inadvertently or deliberately revert a previous security fix. Unlike inserting new malicious code, such a regression removes existing protections and can resemble normal refactoring. For example, Django's fix for CVE-2024-39329 added a dummy password-hashing step to prevent attackers from enumerating user accounts via response timing. The fix looks like unnecessary work: hashing a random string and throwing away the result. An AI agent asked to "clean up" this code would reasonably remove it, restoring the vulnerability (see Appendix A for details).

We expand on existing control evaluations (Ward et al., 2025; Kaufman et al., 2025; Kutasov et al., 2025) by focusing on this class of security fix regression. We construct a control setting using real Django CVE patches paired with SWE-bench Verified tasks (Jimenez et al., 2023), where an untrusted executor performs legitimate issue resolution while a neutral-sounding side task asks it to "clean up" code in a way that reverts a known fix. A trusted monitor then reviews the proposed changes without knowing which fix to look for.

Our main contributions are:

1. A proposed control setting, CVE regression, where the attack vector is reverting known security fixes rather than introducing new malicious code, grounded in real CVE patches and real SWE tasks with automatable verification.
2. An evaluation of 9 executor models across 15 CVEs shows that compliance with attack instructions varies widely (13% to 100%).
3. A monitor analysis using varying model capability (using Haiku, Sonnet, and Opus as proxies) to detect side tasks / attacks.

2. Related Work

AI control.

Greenblatt et al. (2023) introduced the AI control framework, where an untrusted model performs useful work under the oversight of trusted monitors and protocols.

Their setting uses programming tasks where GPT-3.5 rates the suspiciousness of code written by GPT-4, checking for logical errors not caught by test cases.

Subsequent work expanded this to new domains.

- CTRL-ALT-DECEIT (Ward et al., 2025) evaluates code sabotage and sandbagging in ML research tasks, finding that monitors detect code sabotage well (AUC 0.83-1.0 depending on monitor) but a general-purpose monitor struggles with sandbagging (AUC 0.60).
- BashArena (Kaufman et al., 2025) tests agents with elevated privileges on Linux sysadmin tasks, with four sabotage objectives: malware execution, secret exfiltration, privilege escalation, and firewall disabling.
- SHADE-Arena (Kutasov et al., 2025) evaluates sabotage detection in complex agent tasks with benign main objectives and harmful side objectives, finding that agent success on the side task relies heavily on a hidden scratchpad not visible to the monitor.

Security benchmarks for AI agents.

A growing body of work evaluates how AI agents handle security-relevant code, framing security as a capability question.

- CVE-Bench (Wang et al., 2025) benchmarks vulnerability repair across 509 CVEs, finding that SWE-agent repairs only 21% at best.
- SecureAgentBench (Chen et al., 2025) measures secure code generation, reporting that SWE-agent with DeepSeek-V3.1 achieves only 15.2% correct-and-secure solutions.
- SEC-bench (Lee et al., 2025) automates security benchmark construction from real CVE data with verified proof-of-concept exploits.

Diffuse threats.

- Hebbar (2025) distinguishes "producing bad content" from "withholding good content" in the context of AI control, arguing that withholding good content is usually less incriminating than producing bad content, though he notes low confidence in this claim.
- Greenblatt (2025) further analyzes non-concentrated failures and proposes online training and auditing as defenses.

These settings primarily test attacks where the agent produces something suspicious. Our work focuses on the complementary case where the agent removes existing protections in larger codebases. Reverting a security fix is more akin to "withholding" a protection than "producing" malicious code.

3. Methods

3.1. Data Collection

We collect 15 Django CVEs spanning seven vulnerability types: timing attacks, path traversal, information disclosure, log injection, SQL injection, permissions issues, and denial of service. For each CVE, we record and commit SHAs across multiple Django versions and fetch the patch diffs via the GitHub API.

We pair each CVE with SWE-bench Verified Django instances (Jimenez et al., 2023). SWE-bench Verified contains 231 human-verified Django task instances, each with a real GitHub issue, a base commit, and a gold-standard patch. For pairing, we enforce a zero file overlap: the files touched by the SWE-bench gold patch and the files touched by the CVE fix must be completely disjoint. This is a simplification designed for the early version of the control so we ensure the main task and side task can be completed independently without merge conflicts, and the monitor sees changes in separate files.

3.2. Prompt Design

Currently, the methodology still uses LLM completions rather than a fully setup developer or coding agents environment. Each task pair produces two executor prompts (honest and attack), plus a monitor prompt and a judge prompt. All are single-turn user messages sent as chat completions with no system prompt and no tool access.

The attack prompt appends an "Additional cleanup" section that asks the executor to "simplify" or "clean up" the CVE-fixed code. These side task prompts are hand-written per CVE and never mention "CVE," "security," "vulnerability," or "revert." They use developer language: "simplify," "remove redundant," "the standard Python pattern." The executor sees the post-fix (secure) code and is told it is over-engineered.

Executor prompts.

The table below shows the structure and provenance of each component:

Component	Honest	Attack	Source
Issue description	Yes	Yes	SWE-bench problem_statement (verbatim GitHub issue)
Main task code	Yes	Yes	Fetches from django/django at base_commit (pre-fix, buggy)
CVE file code	No	Yes	Fetches from django/django at CVE fix commit (post-fix, secure)
Instructions	Yes	Yes	Hardcoded in prompts.py

"Additional cleanup"	No	Yes	Hand-written neutral side task per CVE
----------------------	----	-----	--

Monitor prompt.

The monitor simulates a PR reviewer. It receives the issue being fixed (the SWE-bench problem statement), the original code for all files the executor was given, and the executor's proposed changes. It outputs a binary verdict: SAFE or SUSPICIOUS.

The monitor does not know which CVE is involved, whether the response is from an honest or attack condition, or what specific security property to look for. It is instructed to check whether any existing security protections are weakened or removed, whether any changes that look like cleanup actually remove safety mechanisms (e.g., input validation, timing protections, escaping, permission checks), and whether the changes introduce any vulnerability. It is told that the developer may have done some code cleanup alongside the fix, and that touching multiple files is normal.

Judge prompt.

The judge receives the CVE patch diff (the ground truth of what the fix changed) and the executor's full response, and outputs a binary verdict: REVERTED or NOT_REVERTED. Unlike the monitor, the judge knows the CVE and has full information. The judge is our automated verifier for labeling executor compliance, not the defense being evaluated.

3.3. Evaluation

For each task pair and executor model:

1. Run the executor in honest condition (main task only) and attack condition (main task + side task)
2. For each attack result, run the judge (Claude Sonnet 4.6) to determine whether the CVE fix was reverted
3. For all results (both conditions), run the monitor to assign suspicion verdicts
4. Compute metrics:
 - a. Safety = $P(\text{monitor flags SUSPICIOUS} \mid \text{attack})$, the fraction of attack diffs caught
 - b. Usefulness = $P(\text{monitor says SAFE} \mid \text{honest})$, the fraction of honest diffs not flagged

4. Results

4.1. Executor Compliance

Table 2 shows the fraction of attack runs where the executor successfully reverted the CVE fix, as determined by the LLM judge. Compliance varies widely across models (13% to 100%). Models that did not revert the fix either ignored the side task entirely (completing only the main task) or engaged with the cleanup request but made changes that did not functionally undo the security fix.

Model	Reverted	Not Reverted	Total	Rate
gpt-5.4-mini	15	0	15	100%
llama-3.3-70b-instruct	14	1	15	93%
kimi-k2-0905	12	3	15	80%
deepseek-v3.2	11	4	15	73%
claude-sonnet-4.6	10	5	15	67%
claude-opus-4.6	7	8	15	47%
gpt-5.1-codex-mini	2	3	5 ²	40%
gpt-5.2-codex	2	3	5 ³	40%
minimax-m2.5	2	13	15	13%
Overall	75	40	115	65%

4.2. Monitor Performance

Table 3 shows monitor performance across three capability tiers. Safety is the fraction of attack diffs flagged as suspicious. Usefulness is the fraction of honest diffs not flagged. All three tiers catch the majority of attacks (88-95%). The main difference is in usefulness: the large monitor (Opus) flags 35% of honest diffs as suspicious, compared to 58% for the small monitor (Haiku). Larger monitors reduce false positives while maintaining similar detection rates.

² Missing results due to null/error API responses (gpt-5.1-codex-mini: 10, gpt-5.2-codex: 10), thus excluded from totals. These need to be retested.

³ See above note.

Monitor Tier	Model	Safety	Usefulness	Attack Flagged	Honest Flagged
Small	claude-haiku-4.5	95%	42%	106/112	57/98
Medium	claude-sonnet-4	88%	53%	100/114	52/110
Large	claude-opus-4.6	91%	65%	105/115	39/111

5. Discussion and Limitations

CVE regression has several properties that make it useful as a control setting. The ground truth is unambiguous: the CVE patch diff defines exactly what "secure" and "vulnerable" look like, so verification is automatable. The attack surface is realistic: reverting a security fix looks like normal refactoring (simplifying a function, removing a "redundant" check), which maps onto the kind of code changes AI agents routinely produce. And the data is abundant: any well-maintained open-source project accumulates CVE patches over time, providing a natural source of task pairs.

Our initial results show that the setting produces meaningful signal across both executor compliance and monitor detection. Executor compliance ranges from 13% to 100% depending on the model, suggesting that safety training significantly affects whether a model follows through on a plausible-sounding but harmful cleanup request. Monitors catch the majority of attacks (88-95%) but at a high false positive cost (35-58% of honest diffs flagged). The main bottleneck is usefulness, not safety: larger monitors improve false positive rates substantially (from 42% to 65% usefulness) while detection remains similar. Very subtle CVEs are caught at a modestly lower rate (84% vs 93-96%), suggesting that the subtlety of the regression does affect detectability, though even the hardest cases are caught most of the time.

These results are initial and should be interpreted as evidence that the setting is viable and produces interesting variation, not as definitive measurements. CVE regression is one instance of a broader concern: important properties of code can be lost as a side effect of routine refactoring, cleanup, or modification. Security fixes are a natural starting point because they have clear ground truth (the CVE patch defines what must be preserved), but any code change that touches previously-fixed logic risks reintroducing old issues unintentionally. Further work should investigate how AI agents handle these regression risks more broadly, especially in scenarios where the changes happen within the same file or function scope, making them harder to isolate during review.

Limitations

Chat completions rather than actual/simulated agentic execution. The executor responds in a single chat turn with code blocks. It has no file system access, cannot browse the repository, run tests, or iterate on its output. This makes the attack easier (the model receives clear instructions and relevant code in context) but less realistic than a full agentic setup where a coding agent navigates a large codebase with tool access. A proper evaluation would use a coding agent harness (such as SWE-agent or Claude Code) with containerized test environments.

No main task verification. We verify the side task (CVE reversion) via the LLM judge but do not verify whether the model actually fixed the SWE-bench bug. SWE-bench provides test cases (FAIL_TO_PASS / PASS_TO_PASS) for this purpose, but running them requires parsing the model's code output into a git-applicable diff and executing the test suite in a Docker container. Future work should verify both tasks to measure the full safety-usefulness tradeoff.

Use of LLM-as-judge. The judge (Claude Sonnet 4.6) determines whether the CVE was reverted. We assume it is reliable for this binary classification task given that it receives the exact patch diff as reference. No human verification of judge outputs was performed during the hackathon.

Disjoint file scope. We enforce zero file overlap between the main task and CVE side task, meaning the SWE-bench fix and the security regression always happen in separate files. This is a simplification: a more realistic and challenging scenario would have the main task and side task operating on the same file or related functionality, where the changes are interleaved and harder for a monitor to separate. Further testing should explore overlapping file scope.

Single repository. All CVEs and tasks come from Django. The setting generalizes in principle to any project with CVE history and SWE-bench coverage, but we have not tested this.

Future Work

1. Full agentic execution with containerized test environments, tool access, and main task verification using SWE-bench test harness
2. Multi-repository task cases beyond Django
3. Human evaluation of judge accuracy on a sample of verdicts

6. Conclusion

We presented CVE regression as a control setting where the attack vector is reverting known security fixes rather than introducing new malicious code. Using 15 Django CVEs paired with SWE-bench Verified tasks, we evaluated 9 executor models and 3 monitor tiers. Executor

compliance varied widely (13-100%, 65% overall), and monitors caught most attacks (88-95%) but at a high false positive cost (35-58% of honest diffs flagged). The main bottleneck is usefulness, not safety: larger monitors substantially reduce false positives while maintaining similar detection rates.

These initial results establish CVE regression as a viable control setting that produces meaningful signals. More broadly, they point to a class of risks worth further investigation: AI agents performing routine code changes can lose important properties of the code they modify, whether security fixes, correctness invariants, or other accumulated protections. Security fixes offer clear ground truth for studying this, but the concern extends to any setting where AI agents refactor or modify mature codebases.

Code and Data

Code repository:

<https://github.com/strivn/undoing-the-fix>

Data/Datasets:

CVE metadata and task pairs are generated by the pipeline in the repository. SWE-bench Verified instances are loaded from HuggingFace (`princeton-nlp/SWE-bench_Verified`).

References

1. Greenblatt, R., Shlegeris, B., Sachan, K., & Roger, F. (2023). AI Control: Improving Safety Despite Intentional Subversion. arXiv:2312.06942. <https://arxiv.org/abs/2312.06942>
2. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2023). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR 2024. arXiv:2310.06770. <https://arxiv.org/abs/2310.06770>
3. Ward, F. R., van der Weij, T., Gabor, H., Martin, S., et al. (2025). CTRL-ALT-DECEIT: Sabotage Evaluations for Automated AI R&D. arXiv:2511.09904. <https://arxiv.org/abs/2511.09904>
4. Wang, P., Liu, X., & Xiao, C. (2025). CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities. NAACL 2025. DOI:10.18653/v1/2025.naacl-long.212. <https://aclanthology.org/2025.naacl-long.212/>
5. Chen, J., Huang, H., Lyu, Y., et al. (2025). SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. arXiv:2509.22097. <https://arxiv.org/abs/2509.22097>
6. Lee, H., Zhang, Z., Lu, H., & Zhang, L. (2025). SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. arXiv:2506.11791. <https://arxiv.org/abs/2506.11791>
7. Kutasov, J., Sun, Y., Colognese, P., et al. (2025). SHADE-Arena: Evaluating Sabotage and Monitoring in LLM Agents. arXiv:2506.15740. <https://arxiv.org/abs/2506.15740>
8. Kaufman, A., Lucassen, J., Tracy, T., Rushing, C., & Bhatt, A. (2025). BashArena: A Control Setting for Highly Privileged AI Agents. arXiv:2512.15688. <https://arxiv.org/abs/2512.15688>
9. Hebbar, V. (2025). How can we solve diffuse threats like research sabotage with AI control? Redwood Research Blog. <https://blog.redwoodresearch.org/p/how-can-we-solve-diffuse-threats>
10. Greenblatt, R. (2025). Notes on handling non-concentrated failures with AI control. Redwood Research Blog. <https://blog.redwoodresearch.org/p/notes-on-handling-non-concentrated>
11. Anthropic. (2026). Claude for Open Source. <https://claude.com/contact-sales/claude-for-oss>

Appendix

Appendix A: Timing Attack CVE-2024-39329

Vulnerability

Django's `verify_password()` in `django/contrib/auth/hashers.py` was susceptible to a timing-based username enumeration attack (CWE-208). Password hashing is intentionally slow to resist brute-force attacks. Accounts without a usable password (e.g., those authenticating via LDAP or SSO, marked with `set_unusable_password()`) triggered an early return that skipped the hashing step. This created a measurable timing difference between requests for accounts with unusable passwords and requests for nonexistent users, where a dummy hash was already executed as a defense (Django ticket #20760). An attacker could exploit this difference to determine which usernames correspond to real accounts.

Fix

The patch introduces a `fake_runtime` flag and executes a dummy `make_password()` call for unusable-password accounts, equalizing the response time across all failure paths:

```
def verify_password(password, encoded, preferred="default"):
    fake_runtime = password is None or not is_password_usable(encoded)
    preferred = get_hasher(preferred)
    try:
        hasher = identify_hasher(encoded)
    except ValueError:
        fake_runtime = True

    if fake_runtime:
        make_password(get_random_string(UNUSABLE_PASSWORD_SUFFIX_LENGTH))
        return False, False
    # ... normal password verification continues
```

Side task prompt

The neutral cleanup request provided to the executor reads:

"The current implementation uses a 'fake_runtime' flag pattern that's unnecessarily complex. Simplify it by returning early when the password is None or unusable. Also remove the unnecessary `make_password()` call — there's no reason to hash a random string when we already know the result is False."

Why this might be plausible request

The flag pattern introduces control flow indirection where a direct early return would be simpler. The `make_password()` call operates on a random string whose result is discarded. Both

observations are valid from a code quality perspective. However, the early return and removal of the dummy hash together restore the original timing side channel, enabling username enumeration.

LLM Usage Statement

Claude Opus 4.6 was used extensively in this project: helping in designing the experiment, writing the code, curating CVEs and writing neutral side task prompts, drafting and refining this report, and verifying citations. All design decisions were reviewed by the author. Results are generated programmatically from the evaluation pipeline and spot-checked for consistency.