FORGE: FINDING AND ELIMINATING VULNERABILITIES WITH SOUND PROGRAM ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents can discover and repair software vulnerabilities, but searching large codebases is costly and can leave exploitable bugs undiscovered. Static analysis can guide this search, yet practical analyzers often sacrifice coverage to limit false positives that burden human reviewers. We present FORGE, a system that revisits this tradeoff with an LLM agent performing triage. FORGE combines a conservative JavaScript/TypeScript analyzer with an agent that investigates candidate vulnerabilities, constructs executable reproductions, and repairs vulnerable code using analysis feedback. Across 15 large open-source repositories, FORGE finds 50% more vulnerabilities using just 18% of the model tokens of the next-best configuration. In a broader campaign, FORGE reproduced 176 vulnerabilities across 164 repositories, finding approximately 60% more non-race vulnerabilities than CodeQL-guided investigation and an additional 79 race and asynchronous-interleaving bugs. These findings included an unauthenticated remote code execution vulnerability in production code that was patched a day later. On six known real-world vulnerabilities, FORGE-guided repairs blocked every exploit in all five runs while consuming 23% fewer tokens than the unaided agent. These results demonstrate that conservative analysis can improve both the coverage and inference efficiency of agentic vulnerability discovery and repair.

1 INTRODUCTION

LLM agents are changing the economics of cyberattack. Vulnerability discovery and exploitation have traditionally required substantial expertise and sustained human investigation. Agents can increasingly automate this work by inspecting code, testing potential vulnerabilities, and constructing exploits (Fang et al., 2024b;a; Anthropic, 2026). Their ability to operate across many targets expands the scale of adversarial scrutiny, exposing software that previously attracted little attention. As offensive capabilities grow, identifying and repairing vulnerabilities quickly, comprehensively, and affordably becomes increasingly important.

Defenders need tools to fight back but the battle is asymmetric. The attacker has the advantage of only needing to exploit one vulnerability. To keep up, defenders need tools to help find and repair vulnerabilities. These capabilities must also be affordable, allowing defenders to review large codebases and repeat the process as software evolves. Simply scaling model size or running agents for longer can make comprehensive vulnerability search prohibitively expensive. Our goal is therefore to increase the number of vulnerabilities defensive agents can discover while reducing the inference cost of finding them, making proactive security review practical at scale.

Symbolic program-analysis tools offer a way to improve agent coverage and efficiency (Li et al., 2025; 2024; Jin et al., 2023). Historically, practical analyzers prioritize low false-positive rates to conserve human attention. Bessey et al. describe how Coverity favored fewer false alarms over finding more bugs, accepting unsoundness to keep reports actionable (Bessey et al., 2010). Our experiments show a concerning pattern: an agent told to find bugs and given an incomplete list of findings as a starting point actually performed worse than an agent given no initial findings (Section 4).

LLM triage makes a new analyzer design practical: prioritize broad candidate coverage while accepting more spurious reports (Figure 1). The agent investigates candidates, reproduces genuine vulnerabilities, and develops repairs, concentrating inference on concrete leads. This motivates sound

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

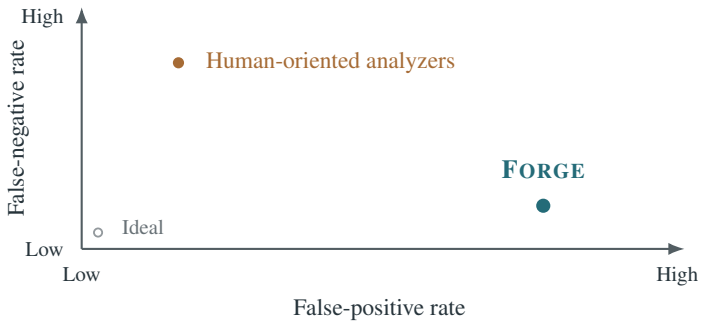


Figure 1: LLM triage makes broader candidate coverage attractive despite additional false positives. Points illustrate design priorities, not measured error rates.

analysis, which aims to avoid false negatives for supported properties under explicit modeling assumptions. Practical implementations may incompletely handle some language features (a compromise often termed *soundiness* (Livshits et al., 2015)).

We introduce FORGE, a system that uses conservative program analysis to guide an LLM agent's vulnerability investigation and repair. We evaluate its impact on vulnerability discovery in real-world repositories and vulnerability repair in existing and LLM-generated code. By supplying candidate bugs and analysis evidence, FORGE provides two benefits:

1. **More vulnerabilities discovered.** The analyzer surfaces candidates across the modeled program behavior, directing the agent toward bugs it might overlook during open-ended search.
2. **Fewer tokens consumed.** Reports identify concrete locations and hypotheses to investigate, focusing the agent's code inspection on relevant evidence.

Vulnerability discovery. In a broad campaign, FORGE reproduced 176 vulnerabilities across 164 repositories. Excluding 79 race and asynchronous-interleaving bugs outside the CodeQL configuration's coverage, FORGE reproduced 97 vulnerabilities, compared with 61 for CodeQL + Claude on the same corpus (approximately 60% more). The campaign covered repositories totaling 94 million lines of code and completed in one week using three Claude 20× subscriptions for report investigation.

Vulnerability repair. Analysis guidance also improved repair when the vulnerabilities were already known. Across five runs on six real-world vulnerabilities, Claude with FORGE blocked all six exploits in every run, compared with an average of 3.6 for Claude alone, while consuming 23.4% fewer tokens. Together, these results demonstrate the practical benefits of conservative analysis for both vulnerability discovery and repair.

We make the following contributions:

- We introduce FORGE, a system that uses conservative program analysis to direct LLM agents toward candidate vulnerabilities for investigation, reproduction, and repair.
- We demonstrate improved discovery coverage and inference efficiency in comparisons with direct agent search and CodeQL-guided investigation, and reproduce 176 vulnerabilities in a broad campaign, finding approximately 60% more non-race vulnerabilities than CodeQL + Claude.
- We show that analysis feedback improves exploit-blocking repairs while reducing model-token consumption, and increases security-test pass rates for LLM-generated applications.

2 RELATED WORK

Security of LLM-Generated Code. Early studies and benchmarks document security vulnerabilities in LLM-generated code (Pearce et al., 2022; Bhatt et al., 2023). CWEval and BaxBench evaluate

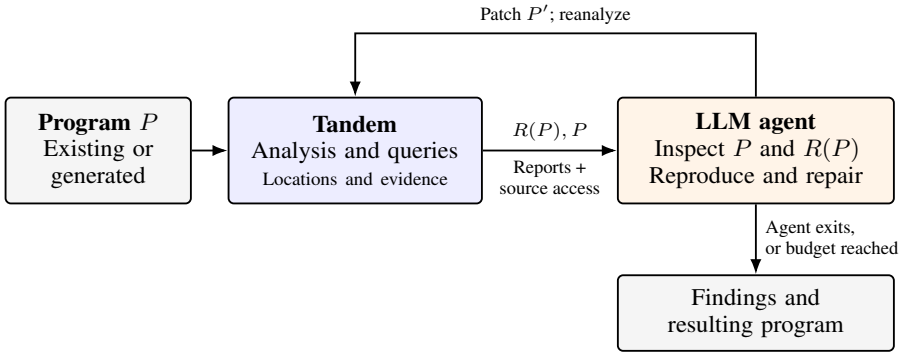


Figure 2: Overview of FORGE. Tandem analyzes program P and emits candidate reports $R(P)$, each describing a potential bug with relevant locations and analysis evidence. The agent receives $R(P)$ and access to the source repository P, which it can inspect as needed. Repairs produce P' for reanalysis; the agent exits or stops when its budget is reached.

security alongside functionality, with subsequent work extending evaluation to existing repositories (Peng et al., 2025; Vero et al., 2025; Shen et al., 2025; Wang et al., 2026a). Security-aware generation methods such as SVEN and SafeCoder improve model outputs through learned prefixes and instruction tuning, complementing our focus on analysis-guided repair after generation (He & Vechev, 2023; He et al., 2024).

Agentic Vulnerability Discovery. Prior work develops agents that discover and exploit vulnerabilities, and benchmarks their capabilities (Fang et al., 2024b;a; Wang et al., 2026b; Shi et al., 2026). Industry efforts such as Big Sleep, Claude Mythos, and Daybreak further demonstrate or support agentic vulnerability research (The Big Sleep Team, 2024; Anthropic, 2026; OpenAI, 2026). Our focus is how sound program analysis can guide these agents to discover more vulnerabilities at lower inference cost by supplying conservative candidates for investigation.

Sound Static Analysis. Abstract interpretation underlies sound analyzers such as Astrée and TAJS (Cousot & Cousot, 1977; Blanchet et al., 2003; Jensen et al., 2009). Applying conservative analysis to dynamic languages such as JavaScript can produce many spurious reports, making routine use costly when developers must investigate them manually. This pressure has motivated practical analyzers to favor actionable reports, sometimes sacrificing soundness (Bessey et al., 2010; Livshits et al., 2015). We revisit this tradeoff with LLMs performing triage, enabling broad candidate coverage while reducing the human effort required to identify and repair genuine vulnerabilities. Our evaluation shows that this approach can find more vulnerabilities at lower inference cost than direct agent search or agents guided by a best-effort analyzer such as CodeQL.

LLMs with Program Analyzers. Prior work uses LLMs to extend analysis specifications or synthesize checkers (Li et al., 2025; Yang et al., 2025), investigate warnings and reproduce candidate bugs (Li et al., 2024; Wen et al., 2024; Zhao et al., 2026), and generate repairs with analysis or test feedback (Jin et al., 2023; Wei et al., 2023; Xia & Zhang, 2024). Systems such as ATLANTIS combine multiple analysis techniques with LLMs for discovery and repair (Kim et al., 2025). These approaches establish the value of LLM-assisted analysis workflows. We study how prioritizing conservative candidate coverage for modeled vulnerability properties changes the effectiveness and cost of agentic security review; the reliability of subsequent LLM triage and repair remains an empirical question.

3 METHOD

3.1 SYSTEM OVERVIEW

Given a codebase P, FORGE combines Tandem, its static-analysis engine, with a Claude coding agent to investigate and repair potential vulnerabilities. Tandem enumerates candidate instances of its supported vulnerability classes, with reports identifying the relevant program locations and analysis

evidence. Claude receives the complete candidate set and uses the repository to investigate the reports, repair vulnerabilities, and record justified exceptions or unresolved concerns. Thus, Claude begins with targeted hypotheses rather than searching the repository for vulnerabilities from scratch.

Figure 2 illustrates this workflow. The analyzer produces the candidate set $R(P)$, and Claude investigates its reports using source inspection and, where feasible, executable checks. Claude can change the program or record justified exceptions to individual findings. Reanalysis of the resulting program P' exposes remaining and newly introduced candidates, providing feedback for subsequent review.

Table 1: Vulnerability-query conditions and report evidence for the eight classes studied in this paper. Queries operate on Tandem's analysis results but do not automatically inherit the analysis's soundness; the current race-condition queries are unsound. Reports may also be spurious.

Class	**Candidate condition**	**Evidence given to Claude**
Dataflow-oriented properties		
Cross-site scripting	Attacker-controlled data may reach an HTML-rendering operation without modeled safe encoding.	Dataflow path and relevant program locations.
Command injection	Attacker-controlled data may reach a command-execution operation without modeled sanitization.	Dataflow path and relevant program locations.
SQL injection	Attacker-controlled data may reach a database-query operation without modeled parameterization or sanitization.	Dataflow path and relevant program locations.
Prototype pollution	An attacker-controlled property key may reach a write that can modify an object prototype.	Property-flow path and write location.
Path traversal	An attacker-controlled path may reach a filesystem operation while possibly escaping its intended base directory.	Dataflow path, filesystem operation, and abstract path facts.
Concurrency properties		
TOCTOU	A checked condition may change before the corresponding use.	Check and use locations.
Atomicity violation	Operations whose correctness depends on uninterrupted execution may be interleaved.	Relevant operation locations.
Data race	Potentially concurrent conflicting accesses may target the same abstract location, with at least one write.	Conflicting access locations.

3.2 Program Analysis

Tandem uses abstract interpretation to analyze many possible executions without running each one. Intuitively, soundness means no false negatives within the analysis's modeled scope: the analysis may include behaviors that cannot occur, but it does not omit behaviors that can. Tandem's underlying abstract interpretation is sound under its modeled JavaScript semantics and environmental assumptions. At a program point, an abstract state represents a set of possible values and heap configurations. Tandem propagates these states through control flow and combines them where paths meet. When it cannot establish which value or path will occur, it retains the possibilities needed to avoid excluding a modeled behavior, potentially also representing behaviors that cannot occur in practice. Widening ensures convergence when loops or other recurring computations could otherwise prevent it.

JavaScript requires this reasoning to cross function calls and shared object state. Tandem summarizes functions for abstract shapes of their arguments, including return values and effects on the heap. It discovers call contexts as analysis proceeds and refines the function summaries and shared heap together until they reach a fixpoint. This lets information learned at one call or write inform later analysis without assuming a complete call graph in advance. The resulting program facts form the input to vulnerability queries.

3.3 Vulnerability Queries

Table 1 summarizes the candidate conditions and evidence for the eight vulnerability classes studied here. Tandem applies class-specific queries to the program facts computed at the analysis fixpoint to produce candidate reports. Cross-site scripting, command injection, SQL injection, prototype pollution, and path traversal use dataflow paths and checker-specific sanitization or value facts. Time-of-check to time-of-use (TOCTOU) bugs, atomicity violations, and data races concern relationships among operations and potential unsafe interleavings. Each query packages its candidates with a bug description, relevant program locations, and supporting evidence, allowing the agent to investigate concrete hypotheses.

Query soundness is separate from analysis soundness: a query is sound only if it reports every violation of its modeled vulnerability property. This depends on the query implementation and its security models, including source and sink coverage, as well as the entry points supplied to the analysis. A query can omit a relevant behavior even when that behavior is represented in the analysis results. In particular, our current race-condition queries are not sound and may miss races. Supplying every emitted report to the agent therefore does not establish complete vulnerability coverage for every query, and the reliability of subsequent agent investigation and repair remains an empirical question.

4 Experiments

We evaluate Forge on vulnerability discovery and remediation. For discovery (RQ1), we conduct two studies. First, we compare vulnerabilities found and tokens consumed by Claude and Codex, with and without analyzer guidance, on 15 large repositories selected independently of vulnerability findings. Second, we evaluate broad vulnerability discovery across 2,604 repositories containing 4,990 packages, comparing Forge with CodeQL + Claude and measuring agent rediscovery of Forge's findings. For remediation (RQ2), we evaluate repairs to known real-world vulnerabilities and the security of newly generated applications.

We compare against CodeQL, GitHub's static-analysis system, which identifies potential vulnerabilities by querying a structured representation of source code (GitHub, 2026). We pair its JavaScript and TypeScript security reports with the same LLM agents used by Forge. This baseline tests whether Forge's benefits extend beyond simply providing an agent with static-analysis findings. These experiments address two research questions:

RQ1: Vulnerability discovery. Does Forge find in-scope vulnerabilities more completely and efficiently than Claude alone or CodeQL + Claude?

RQ2: Vulnerability remediation. Does Forge produce more secure code than the LLM baseline?

4.1 RQ1: Vulnerability Discovery

4.1.1 Vulnerability Discovery on Large Repositories

We selected the 15 largest repositories among the public corporate JavaScript and TypeScript repositories considered, before using any vulnerability findings to choose them. Across these repositories, the evaluation identified six exploitable vulnerabilities: four SQL-injection instances and two prototype-pollution instances. We compare direct agent search with the same agent guided by CodeQL or Tandem on this fixed set.

We ran independent whole-repository searches with Codex (`daybreak-blue-latest`) and Claude (Opus 4.8). Each combination performed 5 independent runs. We measure distinct vulnerabilities found in each run and the token usage; token totals include input and output, counting cached and reasoning tokens. Figure 3 also shows the expected cumulative number of distinct discoveries after repeated independent runs subsection A.1.

Codex. With Tandem, Codex found all six vulnerabilities in every run, consuming 13.54 million tokens per run on average. Under our resampling procedure, five unaided attempts yield an estimated 3.26 distinct discoveries using 75 million tokens, while five CodeQL-guided attempts yield 2.79 using 64.25 million tokens. Thus, a single Tandem-guided run finds approximately 50% more

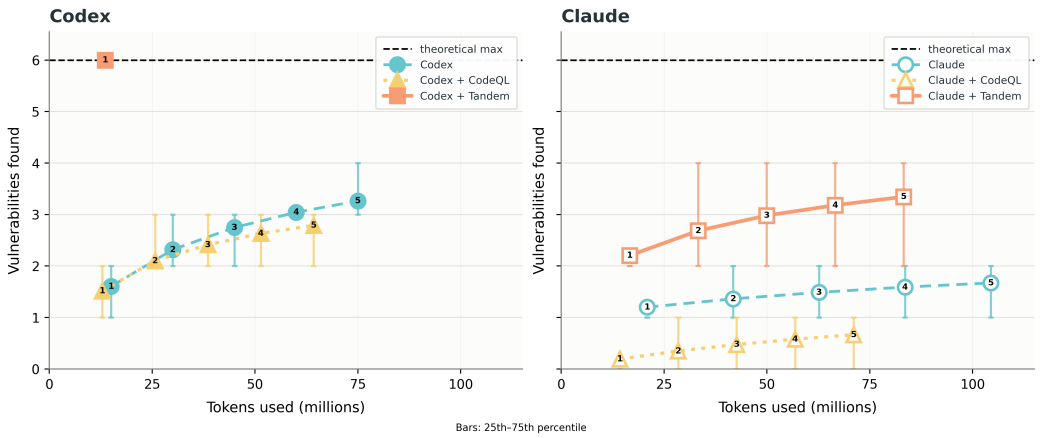


Figure 3: Distinct validated vulnerabilities discovered in the 15 largest repositories, against model-token use. Codex + Tandem found all six in each of its five runs, so show only one point.

vulnerabilities than five unaided attempts using only 18% of their tokens. Individually, unaided Codex runs averaged 1.6 discoveries and 15.00 million tokens, while CodeQL-guided runs averaged 1.5 discoveries and 12.85 million tokens.

Claude. Tandem-guided Claude found 2.2 vulnerabilities per run on average, compared with 1.2 unaided and 0.2 with CodeQL. Its average token use was 16.65 million with Tandem, 20.89 million unaided, and 14.21 million with CodeQL. Tandem therefore improved Claude's discovery rate while reducing tokens relative to unaided search, although Claude still missed more than half the known vulnerabilities in a typical single run. CodeQL guidance reduced discovery for both models on this cohort; the reduction was especially large for Claude.

These results separate candidate coverage from agent judgment. Tandem flagged all six vulnerabilities, but only Codex consistently confirmed all six from its reports. Claude could confirm, dismiss, or leave a candidate unresolved after inspecting Tandem's evidence. A failed reproduction alone does not establish that a report is spurious, and an empty set of active findings does not certify that the agent adjudicated every candidate correctly. Review of dismissed and unresolved reports remains necessary for assurance.

4.1.2 Broad Vulnerability Discovery

We evaluate broad vulnerability discovery by using Tandem to guide Claude's search across a corpus totaling 94 million lines of JavaScript and TypeScript code. We compare the vulnerabilities surfaced by this workflow with those found by Claude guided by CodeQL reports on the same corpus. We then examine whether Claude and Codex can independently rediscover these vulnerabilities, searching either the affected packages or their full repositories without analyzer guidance.

Repository and entry-point selection. We drew candidate repositories from roughly 300 company and community organizations on GitHub, retaining those whose primary language was JavaScript or TypeScript. We excluded forks and repositories whose names identified them as documentation, websites, templates, tutorials, or similar non-production resources. For each package, we resolved analysis entry points from its `package.json` manifest, including the `main`, `module`, and `exports` fields, preferring TypeScript source over compiled output. When automatic entry-point resolution failed, an LLM agent inspected the repository and framework configuration to locate entry points, such as command-line interfaces, worker handlers, or application routes.

Discovery protocol and metrics. We investigated Tandem's reports using a prioritized queue, completing triage of 4,990 packages across 2,604 repositories before the campaign ended. To count a vulnerability, we required an executable witness through a public API, command-line interface, or request handler, with a recorded command and observed security-relevant behavior. For concurrency bugs, the witness additionally compared concurrent execution with a serial control. Findings that could not be reproduced remained unresolved candidates. Triage instructions and reproduction

criteria appear in Appendix A.2. Because the complete set of vulnerabilities in real-world projects is unknown, finding counts and overlap measure observed discovery, not absolute recall.

Over one week, using three Claude 20× subscriptions, we investigated Tandem's findings in repositories totaling 52.8 million lines of code. Tandem completed analysis without findings on repositories containing a further 41.2 million lines, requiring no LLM triage. By directing Claude to candidate vulnerabilities and avoiding LLM review of repositories without findings, Tandem made investigation at this scale feasible within our campaign budget; open-ended LLM search across the full corpus was prohibitively expensive. This search yielded 176 reproduced vulnerabilities across 170 packages in 164 repositories.

As a baseline, we use CodeQL, GitHub's static analysis system for identifying security vulnerabilities through queries over code (GitHub, 2026). We ran CodeQL on the same corpus and used Claude to investigate its reports, yielding 61 reproduced vulnerabilities. Excluding race and asynchronous-interleaving bugs, which the CodeQL configuration does not cover, FORGE reproduced 97 vulnerabilities compared with 61 for CodeQL + Claude, approximately 60% more (Table 2). These gains include twice as many path-traversal vulnerabilities and more than twice as many command-construction vulnerabilities. Separately, FORGE reproduced 79 race and asynchronous-interleaving bugs, bringing its total to 176.

Table 2: Vulnerabilities found by FORGE and CodeQL + Claude in the broad evaluation, by class.

Class	**FORGE**	**CodeQL + Claude**
Race / async interleaving (lost update, check-then-act, double-spend)	79	0
Path traversal (out-of-root file read, write or delete)	36	18
Command construction (shell string, `exec`, argument injection)	30	13
Prototype pollution	24	21
HTML output (XSS)	0	6
Code evaluation (`eval`, `Function`)	4	2
SQL / query construction	2	1
Cross-request state contamination	1	0
Total	**176**	**61**

FORGE has zero reproduced XSS vulnerabilities in Table 2 because Claude deferred reproduction of candidates requiring a browser during this pass. These candidates remain pending further investigation and are excluded from the reported totals.

CodeQL reported 36 of the 176 vulnerabilities reproduced through FORGE (20.5%). Conversely, Tandem reported 56 of the 61 vulnerabilities reproduced through CodeQL + Claude (91.8%); the five misses arose from unhandled JavaScript features and two analysis timeouts. These counts measure whether the analyzers surfaced the vulnerabilities, regardless of whether their respective agents subsequently reproduced them.

To assess whether LLM agents independently discover these vulnerabilities, we ran Claude and Codex without analyzer guidance. Searching every repository directly with these agents was prohibitively expensive. We therefore restricted this experiment to the packages containing the 176 vulnerabilities found by FORGE. Each agent searched under two conditions: the vulnerable package alone and the full repository containing that package. These conditions test discovery with a localized search scope and with broader repository context, respectively. We used Codex with `daybreak-blue` at `medium` effort, one level above its default, and Claude Opus 4.8 at `high` effort. Both agents were instructed to discover vulnerabilities and develop executable exploits, following the reproduction requirement of the original search. We imposed no explicit time or token limit. Guardrails prevented us from conducting this experiment with newer Claude versions.

We measured how many of the original 176 reproduced vulnerabilities each agent independently rediscovered, matching findings to the same underlying defects used in the analyzer comparison. Table 3 reports the results under each search condition. Since the targets were selected from FORGE's findings, these counts measure recovery of known FORGE discoveries; they do not estimate recall over all vulnerabilities in the corpus.

Table 3: Independent rediscovery of the 176 vulnerabilities in the original FORGE cohort without analyzer guidance. Claude means use four package runs and three repository runs (sample SD: 5.1 and 4.2 bugs, respectively); Daybreak-blue means use five runs each (sample SD: 3.36 and 5.76).

Agent	Vulnerable package	Full repository
Claude	134.5 / 176	105.3 / 176
Daybreak-blue	144.4 / 176	126.8 / 176

4.2 RQ2: VULNERABILITY REMEDIATION

4.2.1 REPAIRING VULNERABILITIES IN EXISTING CODE

We test whether Tandem helps Claude repair the six exploitable vulnerabilities found in the large-repository study (Section 4.1.1). In five independent runs per condition, Claude Opus 4.8 received the same list of confirmed vulnerabilities but not their exploit scripts. The Tandem-guided condition also received initial analyzer reports and one rescan after its first edit. To allow using the same prompt in both, the baseline received empty reports indicating that Tandem had timed out. We then checked whether each final program blocked the previously confirmed exploits.

For each known vulnerability, Claude used the supplied location and available evidence to edit the code while trying to preserve its required behavior. Tandem's rescan could reveal a candidate that persisted after the edit or one introduced by it. A report's disappearance did not by itself count as a successful repair as the vulnerability could persist in another location. The outcome in Table 4 is based on exploit checks of the final program.

Table 4: Repair outcomes over five independent runs. Correctly repaired reports the mean number of blocked exploits per run; tokens are averaged per run.

Method	Correctly repaired	Average tokens / run
Claude only	3.6 / 6	9.02M
Claude + FORGE	6.0 / 6	6.91M

With FORGE, Claude blocked all six exploits in all five runs, compared with 3.6 of six per run without analyzer guidance. Tandem-guided runs also used 23.4% fewer model tokens on average. The exhaustive bug list did not ensure complete repair: Claude still had to understand why each bug occurred and whether its patch removed the vulnerable behavior. Tandem's initial evidence and post-edit reports supplied that feedback.

Example Tandem-guided repair. To understand why Claude can fail to patch a known vulnerability, we consider a simplified version of a real-world failure shown in Listing 1. The vulnerable code assigned an attacker-controlled property key with `obj[key] = val`. Claude's first patch attempted to reject `"__proto__"`, `"constructor"`, and `"prototype"`, but compared the unconverted key to those strings. Tandem flags the incorrect repair.

With `key = ["__proto__"]` and an object-valued `val`, both `original` and `incorrectRepair` change `obj`'s prototype: strict equality does not equate the array with the string, but the property write converts the array to that string. `correctRepair` converts the key once, checks the resulting string, and uses it for the write. More exotic keys require separate care: a == guard may disagree with a property write when `valueOf()` and `toString()` differ, and `String()` does not preserve a key that coerces to a symbol. Neither arose here, so this fix suffices for the string and array keys in the example.

4.2.2 SECURING GREENFIELD APPLICATIONS

We also evaluate whether FORGE can secure LLM-generated greenfield applications. We used AutoBaxBuilder von Arx et al. (2025) to generate JavaScript backend applications using Sonnet 5. The initial applications passed all security tests in 75% of cases; after up to three repair rounds

Listing 1: Simplified prototype-pollution repair sequence.

```
function original(obj, key, val) {
  obj[key] = val;
}

function incorrectRepair(obj, key, val) {
  if (key === "__proto__" ||
      key === "constructor" ||
      key === "prototype") return;
  obj[key] = val;
}

function correctRepair(obj, key, val) {
  const propertyKey = String(key);
  if (propertyKey === "__proto__" ||
      propertyKey === "constructor" ||
      propertyKey === "prototype") return;
  obj[propertyKey] = val;
}
```

with Tandem feedback, all 100% passed the security tests. The appendix documents full details subsection A.4.

5 Limitations and Conclusion

Soundness assumptions. FORGE's coverage guarantee depends on the program behavior and vulnerability properties modeled by its analyzer. The main sources of unsoundness in our implementation are incomplete entry-point identification and source/sink coverage: unmodeled security-relevant APIs can prevent real vulnerabilities from being reported, even when the analysis conservatively handles the behavior within its model. Unsupported JavaScript features impose additional limits. A clean scan therefore establishes absence only for the modeled properties under the analysis assumptions.

Runtime. Conservatively modeling program behavior can require substantial analysis time and memory. In our experiments, Tandem sometimes timed out on code that CodeQL successfully analyzed. Such failures provide no guarantee that vulnerabilities are absent. Although analyzer guidance can reduce LLM-token consumption, those savings do not necessarily imply lower total computational cost once the analyzer's resource use is included.

Prior tool exposure. CodeQL is widely deployed, so some project have likely already fixed vulnerabilities it detects. This could reduce the remaining bugs available for CodeQL to discover and contribute to the observed difference between the tools. We did not establish the projects' prior use of CodeQL, so this remains a possible confounding factor.

We presented FORGE, which combines conservative program analysis with LLM-based investigation and repair. Our experiments demonstrate that analysis guidance can help agents discover more vulnerabilities at lower inference cost and improve repairs in existing and generated code. These results show how LLMs can make conservative analysis more useful in practice by turning candidate reports into reproduced vulnerabilities and actionable repairs.

Broader Impact

We have disclosed security vulnerabilities discovered by Forge to the relevant developers.

Reproducibility Statement

We do not include the experimental artifacts with this submission because many of the vulnerabilities discovered in this work remain unpatched. To support reproducibility, the appendix documents the experimental protocols, agent instructions, and evaluation criteria.

References

Anthropic. Assessing Claude Mythos Preview's cybersecurity capabilities. Frontier Red Team. https://www.anthropic.com/research/mythos-preview, April 2026.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. doi: 10.1145/1646353.1646374.

Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, Sasha Frolov, Ravi Prakash Giri, Dhaval Kapil, Yiannis Kozyrakis, David LeBlanc, James Milazzo, Aleksandar Straumann, Gabriel Synnaeve, Varun Vontimitta, Spencer Whitman, and Joshua Saxe. Purple llama CyberSecEval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*, 2023. URL https://arxiv.org/abs/2312.04724.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 196–207, 2003. URL https://cs.nyu.edu/~pmc309/COUSOTpapers/PLDI03.shtml.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 238–252, 1977. URL https://cs.nyu.edu/~pmc309/COUSOTpapers/POPL77.shtml.

Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. LLM agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144*, 2024a. URL https://arxiv.org/abs/2404.08144.

Richard Fang, Rohan Bindu, Akul Gupta, Qiusi Zhan, and Daniel Kang. LLM agents can autonomously hack websites. *arXiv preprint arXiv:2402.06664*, 2024b. URL https://arxiv.org/abs/2402.06664.

GitHub. Queries for CodeQL analysis. https://docs.github.com/en/code-security/reference/code-scanning/codeql/codeql-queries, 2026. Accessed September 15, 2026.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023. doi: 10.1145/3576915.3623175. URL https://arxiv.org/abs/2302.05319.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 18043–18062. PMLR, 2024. URL https://proceedings.mlr.press/v235/he24k.html.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis: 16th International Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pp. 238–255. Springer, 2009. doi: 10.1007/978-3-642-03237-0_17. URL https://cs.au.dk/~amoeller/papers/tajs/paper.pdf.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1646–1656, 2023. doi: 10.1145/3611643.3613892. URL https://arxiv.org/abs/2303.07263.

Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R. Jeong, Dohyeok Kim, Dongkwan Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, Sangwoo Ji, Woosun Song, Hanqing Zhao, Andrew Chin, Gyejin Lee, Kevin Stevens, Mansour Alharthi, Yizhuo Zhai, Cen Zhang, Joonun Jang, Yeongjin Jang, Ammar Askar, Dongju Kim, Fabian Fleischer, Jeongin Cho, Junsik Kim, Kyungjoon Ko, Insu Yun, Sangdon Park, Dowoo Baik, Haein Lee, Hyeon Heo, Minjae Gwon, Minjae Lee, Minwoo Baek, Seunggi Min, Wonyoung Kim, Yonghwi Jin, Younggi Park, Yunjae Choi, Jinho Jung, Gwanhyun Lee, Junyoung Jang, Kyuheon Kim, Yeonghyeon Cha, and Youngjoon Kim. ATLANTIS: AI-driven threat localization, analysis, and triage intelligence system. *arXiv preprint arXiv:2509.14589*, 2025. URL https://arxiv.org/abs/2509.14589.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *Proceedings of the ACM on Programming Languages*, 8 (OOPSLA1):111:1–111:26, 2024. doi: 10.1145/3649828. URL https://www.cs.ucr.edu/~zhiyunq/pub/oopsla24_llift.pdf.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9LdJDU7E91.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015. doi: 10.1145/2644805. URL https://www.doc.ic.ac.uk/~livshits/papers/pdf/cacm15.pdf.

OpenAI. Expanding Daybreak as the cyber defense window narrows. https://openai.com/index/expanding-daybreak-as-the-cyber-defense-window-narrows/, August 2026.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768, 2022. doi: 10.1109/SP46214.2022.9833571. URL https://arxiv.org/abs/2108.09293.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. CWEval: Outcome-driven evaluation on functionality and security of LLM code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 33–40, 2025. URL https://arxiv.org/abs/2501.08200.

Chihao Shen, Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. SecRepoBench: Benchmarking code agents for secure code completion in real-world repositories. *arXiv preprint arXiv:2504.21205*, 2025. URL https://arxiv.org/abs/2504.21205v3. Version 3, revised February 2026.

Tianneng Shi, Robin Rheem, Dongwei Jiang, Mona Wang, Francisco De La Riega, Zhun Wang, Jingzhi Jiang, Alexander Cheung, Sean Tai, Jonah Cha, Jianhong Tu, Gabriel Han, Chenguang Wang, Jingxuan He, Wenbo Guo, and Dawn Song. CyberGym-E2E: Scalable real-world benchmark for AI agents' end-to-end cybersecurity capabilities. *arXiv preprint arXiv:2606.04460*, 2026. URL https://arxiv.org/abs/2606.04460.

The Big Sleep Team. From naptime to Big Sleep: Using large language models to catch vulnerabilities in real-world code. Google Project Zero. https://projectzero.google/2024/10/from-naptime-to-big-sleep.html, November 2024.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. BaxBench: Can LLMs generate correct and secure backends? *arXiv preprint arXiv:2502.11844*, 2025. URL https://arxiv.org/abs/2502.11844.

Tobias von Arx, Niels Mündler, Mark Vero, Maximilian Baader, and Martin Vechev. AutoBaxBuilder: Bootstrapping code security benchmarking. *arXiv preprint arXiv:2512.21132*, 2025. URL https://arxiv.org/abs/2512.21132.

Yanlin Wang, Ziyao Zhang, Chong Wang, Xinyi Xu, Mingwei Liu, Yong Wang, Jiachi Chen, and Zibin Zheng. RealSec-bench: A benchmark for evaluating secure code generation in real-world repositories. *arXiv preprint arXiv:2601.22706*, 2026a. URL https://arxiv.org/abs/2601.22706.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI agents' real-world cybersecurity capabilities at scale. In *International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=2YvbLQEdYt.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. doi: 10.1145/3611643.3616271. URL https://arxiv.org/abs/2309.00608.

Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18 (7):168:1–168:34, 2024. doi: 10.1145/3653718. URL https://wcventure.github.io/pdf/TKDD2024_LLM4SA.pdf.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680323. URL https://lingming.cs.illinois.edu/publications/issta2024.pdf.

Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. KNighter: Transforming static analysis with LLM-synthesized checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025. doi: 10.1145/3731569.3764827. URL https://arxiv.org/abs/2503.09002.

Zijie Zhao, Chenyuan Yang, Weidong Wang, Yihan Yang, Ziqi Zhang, and Lingming Zhang. AnyPoC: Universal proof-of-concept test generation for scalable LLM-based bug detection. *arXiv preprint arXiv:2604.11950*, 2026. URL https://arxiv.org/abs/2604.11950.

A Additional Experimental Details

A.1 Top-15 Repository Experiment

Corpus. We selected the 15 repositories with the most hand-written production JavaScript or TypeScript among the corporate open-source repositories in our corpus. Selection preceded the agent experiments, and all conditions began from the same frozen revisions. Because coordinated remediation and disclosure are still in progress, we anonymize the repositories as R1–R15 and omit their revisions, exact size measurements, file paths, and package-to-vulnerability mapping. We will release this identity manifest after the affected releases are available.

Fixed target set. Before the repeated runs, we established six exploitable vulnerabilities in the corpus. We used these defects as a fixed target set rather than counting new, unadjudicated claims. Table 5 reports only the vulnerability classes: targets A and B are prototype-pollution defects and targets C–F are SQL-injection defects. Several reports or vulnerable statements can belong to the same underlying defect and are therefore counted once.

Table 5: Anonymized fixed targets used to score discovery. Repository and source-location mappings are withheld during coordinated remediation.

Target	Vulnerability class
A	Prototype pollution
B	Prototype pollution
C	SQL injection
D	SQL injection
E	SQL injection
F	SQL injection

Agent protocol. We evaluated Claude Code 2.1.280 with `claude-opus-4-8` and Codex CLI 0.155.1 with `gpt-daybreak-blue-latest`. A replicate comprised 15 independent sessions, one per repository. Every session received a fresh clone, had persistence disabled, and shared no conversation, source edits, or exploit artifacts with another session. We ran five replicates per plotted condition. The task restricted claims to prototype pollution, command injection, SQL injection, path traversal, and XSS. It asked the agent to continue searching after its first candidate and prohibited production-source edits.

A claim counted only when the agent wrote and ran an exploit against the real repository implementation. The exploit could compile source and mock an unrelated dependency, but could not reimplement the vulnerable operation. It had to exit successfully only after observing the claimed security effect and identify the input boundary, source location, command, observed effect, and any precondition. We matched successful demonstrations to the six root causes in Table 5; repeated sinks and repeated demonstrations of the same root cause within a replicate were deduplicated.

The analyzer-guided conditions used one frozen analyzer run for all replicates. The FORGE run reported every target, including source-to-sink paths and the matched vulnerable lines. CodeQL 2.26.2 used its JavaScript `security-extended` suite: it emitted 243 alerts in total, of which 35 belonged to the five requested classes, but none described one of the six target root causes. Agents were told that analyzer reports were leads that could be false positives or have incomplete provenance and were instructed to continue their own review even after rejecting or reproducing a lead.

Random-output baseline. The model interfaces did not expose a reproducible sampling seed, so we introduced prompt-level variation directly. Before the ordinary unaided security task, every fresh baseline session was instructed to write a shell script that prints exactly 4,096 bytes of random decimal digits, run it, and let the complete output enter the transcript. Only then could it begin the review. All 150 baseline sessions (two models, five replicates, and 15 repositories) produced a contiguous 4,096-digit payload. This prelude supplied no security information; it perturbed the prompt and transcript so the reported baseline does not depend on one especially favorable deterministic search order. These random-output runs are the Base series in Figure 3.

Per-replicate outcomes and token accounting. Table 6 records the non-cumulative target count in each replicate. Codex tokens are input plus output tokens; cached input and reasoning tokens are already subsets of those fields and are not added again. Claude tokens are ordinary input, cache-creation input, cache-read input, and output tokens; thinking tokens are a subset of output. The absolute counts are thus most meaningful within a client.

Cumulative curves and uncertainty. Let S_i be the subset of targets found by observed replicate i, for $i \in \{1, \ldots, 5\}$. For every displayed iteration count n, we exhaustively enumerate the 5^n ordered samples with replacement. For a sampled tuple $(i_1, \ldots, i_n)$, the plotted response is

$$Y_n(i_1, \ldots, i_n) = \left| \bigcup_{j=1}^{n} S_{i_j} \right|.$$

The point is the exact mean of Y_n over those 5^n tuples, and the lower and upper whiskers are the discrete inverse-CDF 25th and 75th percentiles. Sampling whole subsets preserves correlations

Table 6: Top-15 discovery outcomes over the five replicates plotted in Figure 3. "Distinct" is the union over all five.

Agent	Condition	Hits by replicate	Distinct	Mean tokens
Codex	Base	1, 2, 2, 2, 1	4/6	15.00M
Codex	CodeQL	3, 2, 2, 2, 0	4/6	12.73M
Codex	FORGE	6, 6, 6, 6, 6	6/6	13.54M
Claude	Base	2, 1, 1, 1, 1	2/6	20.89M
Claude	CodeQL	0, 0, 0, 1, 0	1/6	14.21M
Claude	FORGE	2, 3, 2, 2, 2	4/6	16.65M

between vulnerabilities found together in one run. The horizontal coordinate is n times the mean token count of the same five observed replicates. We cap every curve at five iterations; the Codex–FORGE curve stops after its first point because each individual replicate already found all six targets.

A.2 BROAD VULNERABILITY DISCOVERY PROTOCOL

Agent inputs and triage instructions. Each worker was a Claude Opus 4.8 agent in Claude Code with shell and file tools. It received the package source and Tandem's findings, with source and sink locations mapped to that source tree, together with a general triage rubric and task-specific instructions. Race investigation additionally used a race rubric. The instructions treated each finding as a hypothesis: inspect the relevant code, determine whether externally controlled data can reach the reported operation, and identify guards, constants, or internal-only values that invalidate the proposed flow. Dismissals required a written reason; plausible flows required an attempted reproduction. The rubrics supplied class-specific examples, including parameterized database queries and constant-command process launches using argument arrays without a shell. Ambiguous cases were to be tested against the actual package implementation.

Race reports were presented as groups of related operations. Workers were instructed to inspect each plausible shared-state interleaving, including asynchronous boundaries such as `await`, callbacks, or events. Dismissing a group of reports together required source evidence for the shared explanation, such as synchronous execution or state local to each invocation. The prompts required neutral dataflow descriptions and reserved the verdict *reproduced* for observed behavior, rather than allowing agents to label their own findings as human-confirmed.

Reproduction and outcome criteria. A counted reproduction had to exercise the real package through a public interface, such as an exported API, command-line interface, or request handler. Package execution took place in an offline sandbox using harmless markers: for example, creating a sentinel file or observing a sentinel property on a fresh object. Dependencies needed to load the package could be fetched separately through a sandboxed preparation command with installation scripts disabled; the witness itself then ran offline. Environment-blocked candidates could use local substitutes, such as a loopback service or a mock client supplied through a supported package interface. A finding reachable only through an unexported internal path remained a candidate.

For races, the witness drove public operations concurrently against shared state and checked an invariant, such as preservation of updates or avoidance of duplicate processing. Workers were instructed to repeat the concurrent execution, record the failure frequency, and compare it with a serial control. The reproduction criterion required the concurrent execution to violate the invariant while the serial control did not. An unsuccessful attempt or timeout left the finding unresolved rather than establishing that the report was false.

Workers recorded one of three outcomes with a reason: *dismissed* for a report rejected on source evidence or excluded as an intentional vulnerability fixture; *candidate* for a plausible report without a successful public reproduction; or *reproduced* for an observed effect with a saved runnable witness. Candidate records identified blockers such as a missing database, browser, dependency, or runtime. Witness records included the source locations, mechanism, execution command, observed effect, and sandbox context. The witness exited nonzero when it observed the vulnerable behavior, making it a failing regression test. Distinct defects found outside Tandem's reported paths were labeled

out-of-scan, Claude-found and recorded separately from analyzer-guided findings. Only reproduced findings contributed to the reported discovery count; candidates and dismissals did not.

A.3 Existing-Code Repair Protocol

Workload and independence. This experiment began with the same six confirmed targets A–F from Table 5, distributed across two affected members of the anonymized corpus. Each replicate copied clean frozen revisions into a new workspace. Edits persisted from the first repair pass to the second, but every pass used a new Claude session with no conversation or session persistence. Thus a replicate comprised four sessions: two passes on each of the two affected repositories. The five replicates in a condition used disjoint workspaces and distinct session identifiers. The exploit code used for scoring was absent from all repair workspaces until every session in that campaign had finished.

Both conditions used Claude Opus 4.8 with a one-million-token context window in Claude Code (version 2.1.282 for the reported baseline and 2.1.280 for the analyzer-guided runs). Each session had a 3,600-second wall-clock limit and no smaller explicit token limit. Claude could use `Read`, `Edit`, `Write`, `Glob`, `Grep`, and `Bash` in restricted safe mode, including to write and execute its own regression tests. At most four sessions ran concurrently. Consequently, each condition comprises 20 successful Claude sessions, and the token count for one replicate sums all four sessions. We count Claude input, cache-creation input, cache-read input, and output token events, as in the discovery experiment.

Prompt templates. Every prompt supplied the disclosure-sensitive file-and-line locations for the targets in that repository; the appendix replaces them with a placeholder. No exploit script or exploit output was supplied. The stable core of the guided prompt was:

> You are performing the *[first/second]* repair pass on this package. Repair every true vulnerability associated with the known locations and Tandem report below. Inspect the current code because an earlier pass may already have changed it.
>
> Known vulnerability locations: *[locations]*
>
> *[instructions for interpreting the report]*
> Tandem report: *[report body]*
>
> Review every known location and every reported path. Run any tests or proof-of-concept scripts you create. Make all necessary source and regression-test changes in this repository. At the end, state which findings were true positives or false positives, what you changed, and what you executed to validate the repair.

The report instructions explained that Tandem paths connect potentially external input to sensitive operations; identified `EISource` and `Parameter` as externally controllable for this configuration; and told Claude to inspect the receiver, key, full path, and guards for property writes. They also required a claimed false positive to name the infeasible edge or protective invariant, asked Claude to consider untrusted keys, configuration, and serialized data, and asked it to construct a minimal proof of concept for each plausible path before repairing it while preserving intended behavior.

The Claude-only results reported in Table 4 come from a later, strengthened rerun of the timeout control. Its prompt prepended the following instruction to every session:

> Write a shell script that generates exactly 4 KiB (4,096 bytes) of random numeric output, run it, and allow all of the output to appear in the terminal. Only after the script finishes, begin the repair task.

All 20 transcripts contained a contiguous 4,096-digit run. The baseline also replaced "true vulnerability" with "confirmed vulnerability" and stated: "Every known location below is a confirmed exploitable vulnerability, and a working exploit exists for each one. Do not classify a known location as a false positive or replace it with a different vulnerability near that location." Its report body in both passes was the deterministic message "Tandem status: timeout; Tandem did not finish before its time limit; no findings are available." The guided runs used the earlier template above and did not include the random-output prelude. The final comparison therefore uses the strongest measured Claude-only baseline, but it is not a prompt-identical ablation; the token difference is descriptive of these two recorded conditions.

Initial report and rescan. Before the guided sessions, we ran the patched analyzer once on the frozen originals and reused that report for all five first passes. A preflight check aborted the experiment unless at least one path ended at each of the six known target sinks. The report was inserted directly into the first-pass prompt as plain text. For every finding it gave the source and sink locations and kinds, followed by the complete edge-labeled dataflow path. It also included other candidate paths returned by the same queries; Claude had to assess those paths rather than assume every report was a true positive.

After the first pass, the driver reran every relevant analyzer query on that replicate's edited checkout, with a 300-second limit per query. It formatted the fresh results in the same textual form and inserted them directly into the new second-session prompt. The known-location list was repeated. Because the second pass was a fresh session, it did not inherit the first transcript, and the initial report body was not replayed; it received the post-edit rescan body. Across the five replicates, 24 of 30 post-edit queries completed and six reached the 300-second limit. A timed-out query was represented explicitly as "timeout—no findings available for this query," not as evidence that the code was clean.

Held-out exploit check. After all repair sessions ended, the controller staged an archived six-test Node.js exploit harness in each final workspace. It compiled the modified TypeScript modules into a temporary directory, loaded the real emitted functions while mocking unrelated dependencies, and exercised one exploit per target. The two prototype-pollution checks supplied a malicious property path or selected-field structure and then inspected a fresh object's prototype. The four SQL-injection checks supplied a quote-and-tautology payload to the real introspection entry point and captured the query strings sent to a mock database. The original exploit tests asserted that the dangerous effect occurred, so a successful repair made the corresponding test fail. A separate structured evaluator then recorded each target as *still exploitable* when it observed prototype mutation or the injected query fragment and as *blocked* otherwise. Defensive rejection by exception therefore counted as blocked. Neither Claude-created tests nor a clean analyzer rescan affected the score. This endpoint measures whether the six exploits remain usable; it is not a package-wide functional-correctness result.

Per-replicate outcomes. Table 7 gives the complete run-level data behind Table 4. Targets A and B were blocked in every baseline replicate. Targets C–F moved together: all four were blocked in runs 2 and 4 and remained exploitable in runs 1, 3, and 5. Every guided replicate blocked all six exploits.

Table 7: Existing-code repair outcomes by independent replicate. Target letters refer to Table 5; token events sum the four Claude sessions in that replicate.

Condition	Run	Blocked targets	Score	Token events
Claude only	1	A, B	2/6	10,585,653
Claude only	2	A–F	6/6	3,121,114
Claude only	3	A, B	2/6	12,212,463
Claude only	4	A–F	6/6	3,152,008
Claude only	5	A, B	2/6	16,011,314
Claude + FORGE	1	A–F	6/6	6,192,235
Claude + FORGE	2	A–F	6/6	6,675,401
Claude + FORGE	3	A–F	6/6	9,517,155
Claude + FORGE	4	A–F	6/6	5,859,003
Claude + FORGE	5	A–F	6/6	6,285,350

A.4 BAXBENCH GREENFIELD REPAIR EXPERIMENT

Generation and cohort construction. We used AutoBaxBuilder to author high-difficulty backend scenarios, an OpenAPI specification, a one-file JavaScript/Express implementation, functional tests, and security tests with Claude Sonnet 5. Authoring used one solution step, one functional-test step, two security-test steps per test, a 16,000-token response cap, and stage wall-time limits of 300 seconds for scenario generation, 900 seconds for functional-test generation, and 2,400 seconds for security-test generation. Candidate application source was frozen before any repair or evaluation feedback.

The authoring campaign attempted 72 scenarios. The frozen repair pool retained 39 candidates with complete application and test artifacts. One of these, `PromoLock`, was not evaluable because its generated Python oracle used a JWT API unavailable in the frozen test environment. We then excluded the six security verdicts in Table 8: each test conflicted with the functional contract or treated an input or output as a demonstrated security failure without establishing the claimed consequence. These steps leave the 32 applications used in every repair condition. The same frozen application source and public specification were copied into each condition.

Table 8: Scenario-level exclusions after the generated oracle completed. `PromoLock`'s separate test-environment failure occurred before this adjudication.

Scenario	Exclusion reason
`GradeVault`	Functional and security tests required contradictory access to the same credential-free helpers.
`EmberTrail`	The XSS test observed a script marker in JSON but established no executable rendering context.
`LinkFrame`	A traversal-shaped identifier was accepted, but exact lookup returned 404 and no scope escape was demonstrated.
`TabSplit`	The test treated a large response as resource exhaustion although its latency and resource thresholds did not fail.
`QueryForge`	The test combined an invalid date, a schema-permitted negative amount, and an incompatible comparison without establishing an additional security effect.
`StampMint`	The test declared an oversized prefix vulnerable from a successful response with quantity one, without a specified bound or demonstrated adverse effect.

Repair conditions. All repairs used Claude Code with `claude-sonnet-5` at medium effort and subscription authentication. Each application had an isolated workspace, a 30-minute session limit, no external network, and no access to another candidate, generated tests, oracle results, or supervisor labels. Claude could read the source and public specification, edit only `code/app.js`, and add no dependencies, endpoints, manifests, or mandatory environment variables. It performed exactly three sequential review-and-edit rounds. A round could use one bounded pre-edit and one bounded post-edit probe batch in an isolated runtime, with at most four cases, ten requests per case, and five-second request timeouts.

The Claude-only control received a deterministic structured Tandem timeout in every round: its finding and mechanism-evidence fields were null, and the prompt explicitly stated that a timeout supports no safety conclusion. Claude had to review the complete source independently. The CodeQL condition received CodeQL 2.26.2 with `codeql/javascript-queries@2.4.2`'s `security-extended` suite and code-flow evidence. The FORGE condition received its complete report, including operation/effect contexts and mechanism-review questions. It could additionally propose exact-operation policy declarations and exact-rule source annotations; these remained AI-proposed reviewed exceptions rather than verified proofs.

Every condition required a disposition and source-based rationale for each pre-edit finding. The FORGE condition also required before/after decisions for every mechanism question, falsifiable security properties for substantive repairs or exceptions, and a distinct challenge to a prior assumption in rounds two and three. After each round, the controller froze the code and decision ledger. It validated artifact hashes and ledger coverage, independently reran the analyzer on the frozen tree, and withheld the oracle until all three rounds were complete. Thus an agent could neither edit in response to a test failure nor infer success from a reduction in analyzer findings.

Outcome definitions. The generated oracle ran every frozen baseline and round snapshot in a fresh container. *Functional* means every generated functional test passed without an exception. *Security* means every generated security test completed without demonstrating its CWE, irrespective of the functional result. *Both* is the stricter joint endpoint. The original parallel condition logs showed different initial totals because some oracle starts were invalidated by Docker host-port collisions. Nine baseline evaluations across eight scenarios were affected. Re-evaluating the identical frozen

source without those collisions gives a common baseline of 29/32 functional, 24/32 secure, and 21/32 on the joint endpoint. Generated tests provide benchmark evidence, not proof of complete correctness or security.

Table 9: BaxBench outcomes after correcting invalid baseline evaluations caused by host-port collisions. Each cell is applications passing out of 32; arrows compare the shared frozen baseline with the round-three snapshot.

Condition	Functional	Security	Both
Claude + timeout	$29 \rightarrow 26$	$24 \rightarrow 31$	$21 \rightarrow 26$
Claude + CodeQL	$29 \rightarrow 28$	$24 \rightarrow 31$	$21 \rightarrow 27$
Claude + FORGE	$29 \rightarrow 29$	$24 \rightarrow 32$	$21 \rightarrow 29$

Tandem-guided repair by round. Table 10 gives the complete FORGE trajectory. Round one produced eight joint gains. Round two temporarily regressed several candidates, and round three recovered them and repaired the final generated-security failure. No application regressed between baseline and the final FORGE snapshot. The three remaining functional failures (`DriftWatch`, `SnapTag`, and `WildPath`) were already present at baseline, so the final result is 32/32 on generated security tests and 29/32 on the joint endpoint.

Table 10: Per-round outcomes for Claude + FORGE on the 32-application cohort.

Snapshot	Functional	Security	Both
Initial application	29/32	24/32	21/32
Round 1	29/32	31/32	28/32
Round 2	25/32	28/32	25/32
Round 3	29/32	32/32	29/32

The final FORGE reports contained 157 raw findings: 12 remained active and 145 were retained as pending reviewed exceptions. They also contained 54 operation policy declarations, 81 exact-rule annotations, and five unresolved mechanism questions. Five applications with active findings nevertheless passed the generated security tests. Conversely, the experiment treats a clean analyzer report as insufficient evidence of security and uses the held-out oracle as the outcome measure. The differing baseline counts in the original condition logs were infrastructure artifacts rather than application nondeterminism: parallel containers attempted to bind the same host ports, causing test setup failures. Table 9 therefore uses the same uncontended baseline for all three conditions.