SABER: FROM VULNERABILITY REPORTS TO EXECUTABLE REPAIR TASKS AT SCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

Training reliable vulnerability-repair agents benefits from executable tasks that verify both security and functionality. However, existing resources remain limited in scale or executability. We introduce SABER (**S**calable **A**gent-**B**ased **E**xecutable **R**epair), a pipeline that turns public vulnerability evidence into repository-level repair tasks, each with a reproducible environment, an upstream fix, and behavioral tests. In particular, SABER first links NVD and GHSA records into deduplicated targets, whose complementary fields fill gaps left by either source alone. A tool-using agent then interprets each target's irregular evidence and synthesizes a candidate repair task. It can adapt to heterogeneous project contexts without project-specific rules or manual reconstruction. Quality control further involves deterministic processing, layered validation, and feedback-driven refinement. Applied to 60,000 targets, SABER promotes 22,089 tasks across 10,163 upstream repositories and more than 60 languages. Sampled ablations demonstrate the benefits of combined evidence, complementary validation gates, and two-layer trajectory filtering. Fine-tuning Qwen3.5-4B and Qwen3.5-35B-A3B on filtered SABER trajectories improves vulnerability-repair performance across multiple programming languages and evaluation criteria. Gains reach 7.83/6.96 points on PatchEval Strict, 14.33/5.00 on SEC-bench Strict, and 11.63/10.85 on Vul4J.

1 INTRODUCTION

Software vulnerabilities are routinely discovered in widely used projects. Unremediated flaws can expose downstream systems to compromise (National Institute of Standards and Technology, 2026b; GitHub, 2026). As large language models (LLMs) continue to advance, LLM-powered agents are becoming capable of discovering and exploiting software vulnerabilities, increasing the urgency of timely remediation (Hu et al., 2025). Meanwhile, their growing repository-level software-engineering capabilities make automated vulnerability repair increasingly feasible. For such repair to be reliable, it must eliminate vulnerable behavior while preserving intended functionality and project buildability. Developing and evaluating agents against this objective therefore calls for large collections of executable vulnerability-repair tasks with reproducible environments and verifiable behavioral checks.

Several vulnerability-repair benchmarks have been released in recent years (Lee et al., 2025; Shi et al., 2026). Large-scale datasets associate vulnerabilities with repositories, commits, or functions, but omit reproducible historical environments and executable repair checks (Bhandari et al., 2021; Akhoundali et al., 2024; Ding et al., 2025). ARVO and CyberForge provide dockerized environments with executable vulnerability checks, but cover only 6,138 and 1,034 C/C++ OSS-Fuzz instances, respectively. Moreover, ARVO is restricted to crash-manifesting cases with one trigger each, leaving functionality regressions unchecked. CyberForge uses synthetically injected rather than real vulnerabilities (Mei et al., 2026; Lbath et al., 2026). Curated repair benchmarks support richer security and functionality checks through manual instance-level reconstruction, resulting in small collections (Wei et al., 2025; Yu et al., 2026; Bui et al., 2026). Consequently, scalable construction of authentic, executable, and behaviorally verifiable vulnerability-repair tasks remains unresolved. This limits access to high-quality trajectories for training vulnerability-repair models.

The emergence of increasingly capable tool-using coding agents makes this challenge more tractable. By navigating repositories, invoking tools, running tests, and revising code based

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.

Table 1: **Comparison of representative vulnerability-repair datasets with SABER.** Sources for prior datasets, in row order: (Fan et al., 2020), (Reis & Abreu, 2021), (Bhandari et al., 2021), (Chen et al., 2023), (Akhoundali et al., 2024), (Ding et al., 2025), (Mei et al., 2026), and (Lbath et al., 2026). ✓, ×, and ◯ indicate yes, no, and partial support. `Real vul.`: real-project rather than intentionally injected vulnerabilities. `Multi-lang`: multiple programming languages. C/C++-only collections are ×. `Patch`: repository-level upstream fix. ◯ denotes function-level before/after pairs only. `Env.`: directly executable repository-level environment. ◯ denotes repository-level source with no executable environment. `Vuln.Verif.`: executable vulnerability reproduction and post-repair elimination. `Reg.Tests`: tests for repair-induced functionality regressions.

Work	Scale	Real vul.	Multi-lang	Patch	Env.	Vuln.Verif.	Reg.Tests
Big-Vul	3,754 vul.; 348 proj.	✓	×	✓	×	×	×
Security-patch	5,942 patches; 1,339 proj.	✓	✓	✓	◯	×	×
CVEfixes	5,365 vul.; 1,754 proj.	✓	✓	✓	×	×	×
DiverseVul	7,514 commits	✓	×	×	×	×	×
MoreFixes	26,617 vul.; 6,945 proj.	✓	✓	✓	×	×	×
PrimeVul	6,827 commits; 755 proj.	✓	×	◯	×	×	×
ARVO	6,138 vul.; 311 proj.	✓	×	✓	✓	✓	×
CyberForge	1,034 vul.; 80 proj.	×	×	✓	✓	✓	✓
SABER	***22,089 vul.; 10,163 proj.***	✓	✓	✓	✓	✓	✓

on execution feedback, these agents support long-horizon, repository-scale coding and self-verification (OpenAI, 2026; Anthropic, 2026; Z.ai, 2026; Kimi Team, 2026). For example, SWE-smith and SWE-rebench V2 demonstrate that agents can construct reproducible environments and executable software-engineering tasks across projects and languages (Yang et al., 2025; Badertdinov et al., 2026). Nevertheless, vulnerability-repair task construction poses additional challenges beyond general software-engineering task generation. In particular, each task must identify the correct upstream repository and vulnerable revision while reproducing the reported security behavior. It must also ground its reference repair in a real upstream fix and include regression checks without exposing hidden artifacts. These requirements motivate structural, provenance, semantic, and execution validation.

To address this gap, we introduce SABER (**S**calable **A**gent-**B**ased **E**xecutable **R**epair), an agent-driven pipeline for constructing executable vulnerability-repair tasks from public vulnerability evidence. SABER first constructs deduplicated evidence targets by linking NVD (National Institute of Standards and Technology, 2026b) and GHSA (GitHub, 2026) records. Their complementary evidence expands the set of constructible targets by filling synthesis-critical gaps left by either source alone. It also strengthens the evidential grounding of each task. For each target, SABER employs a tool-using construction agent to interpret irregular evidence and adapt the construction process to heterogeneous project contexts. Together, these components support construction across languages and projects without project-specific rules or manual reconstruction, while enabling security and functionality checks for each task. Flexible agent-based synthesis can introduce structural, provenance, semantic, and execution inconsistencies. We address these through deterministic postprocessing followed by layered validation of the resulting artifacts. Correctable failures are repaired using structured feedback and revalidated. Only candidates that pass the complete validation stack are promoted. As summarized in Table 1, the resulting dataset exceeds prior executable repair datasets in scale. It also combines multi-language coverage, repository-level executability, upstream patches, executable vulnerability verification, and regression testing. Together, these properties enable collection of high-quality, execution-verified trajectories for vulnerability-repair training.

Our data production experiments demonstrate that SABER constructs vulnerability-repair tasks at substantial scale and diversity. Ablations suggest that each component of the full evidence bundle contributes to its effectiveness. Validation analysis shows that the four gates make complementary contributions to task quality. A three-round repair budget preserves task yield without excessive retries. Filtering ablations show that combining PoC-based and hidden tests yields higher-quality training trajectories and better model performance. Runtime isolation further neutralizes the vast majority of observed solver-evasion attempts. Fine-tuning Qwen3.5-4B and Qwen3.5-35B-A3B on filtered trajectories further improves performance on modern vulnerability-repair benchmarks. For

4B and 35B-A3B, respectively, PatchEval Strict improves by 7.83 and 6.96 points, SEC-bench Strict by 14.33 and 5.00 points, and Vul4J by 11.63 and 10.85 points.

Our contributions are fourfold:

- **Complementary evidence integration.** To our knowledge, we are the first to integrate linked NVD and GHSA records with crawled references. This expands synthesis coverage while strengthening the evidential grounding of constructed tasks.
- **Agent-based task synthesis.** We introduce a language-flexible construction process in which a tool-using agent interprets diverse, irregular evidence and adapts to heterogeneous project contexts. This enables task construction beyond patterns anticipated by fixed rules and supports more comprehensive verification.
- **Layered quality assurance.** We combine deterministic postprocessing with structural, provenance, semantic, and execution validation. Together, these mechanisms assess candidates for evidence support, behavioral fidelity, and executability.
- **Scalable construction.** We demonstrate scalable construction of a large, multi-language dataset of repository-level vulnerability-repair tasks with execution and validation contracts. Experiments demonstrate their effectiveness for trajectory filtering and model training. Our produced SABER tasks will be made publicly available.

2 Related Work

2.1 Vulnerability Data Synthesis

Research on vulnerability-data synthesis has evolved substantially. Early work focused on mining static links between vulnerabilities and code changes, while recent efforts reconstruct execution contexts to reproduce vulnerabilities and verify repairs.

Big-Vul associates C/C++ CVEs with repositories, commits, and functions (Fan et al., 2020). The security-patch dataset and CVEfixes extend such collections to multiple languages at the repository and commit levels (Reis & Abreu, 2021; Bhandari et al., 2021). DiverseVul, MoreFixes, and PrimeVul further expand coverage and improve repository discovery, labeling, and deduplication (Chen et al., 2023; Akhoundali et al., 2024; Ding et al., 2025). These collections nevertheless remain centered on code changes or individual functions. Reproducible historical dependencies and build configurations are often missing. Without this execution context, vulnerable behavior may not be reproducible, and candidate repairs cannot be reliably verified. ARVO and CyberForge move closer to execution-grounded data through dockerized environments and executable vulnerability checks. However, both remain limited in scale and specific to C/C++ OSS-Fuzz projects. Additionally, ARVO evaluates each vulnerability using only a single PoC-based check, whereas CyberForge uses synthetically injected vulnerabilities (Mei et al., 2026; Lbath et al., 2026).

Vulnerability-repair benchmarks instead rely on instance-level curation. Vul4J and Vul4C manually reconstruct executable vulnerabilities in Java and C/C++, respectively, with verification based on PoV tests or exploits (Bui et al., 2022; Hu et al., 2025). PatchEval builds security and functionality sandboxes for 230 of 1,000 multilingual CVEs (Wei et al., 2025). Vul4Py packages paired exploits with project-native functional tests in pinned Python environments (Bui et al., 2026). Such high-fidelity evaluation typically requires substantial manual effort for environment reconstruction, vulnerability reproduction, and repair verification. Collections built this way are often small, and their curation workflows are difficult to scale through full automation.

For training-oriented vulnerability-repair data, broad coverage across repositories, languages, and real-world vulnerabilities is important. Reproducible execution and moderate verification are also needed to ensure task executability and support trajectory filtering, while balancing verification coverage and training-data retention. Our work bridges these two lines by constructing a large, multi-language dataset of repository-level vulnerability-repair tasks. Each promoted task includes a reproducible environment, an evidence-grounded source repair, executable vulnerability verification, and targeted functionality checks. The dataset thus combines the scale of static collections with executable verification for training-data construction.

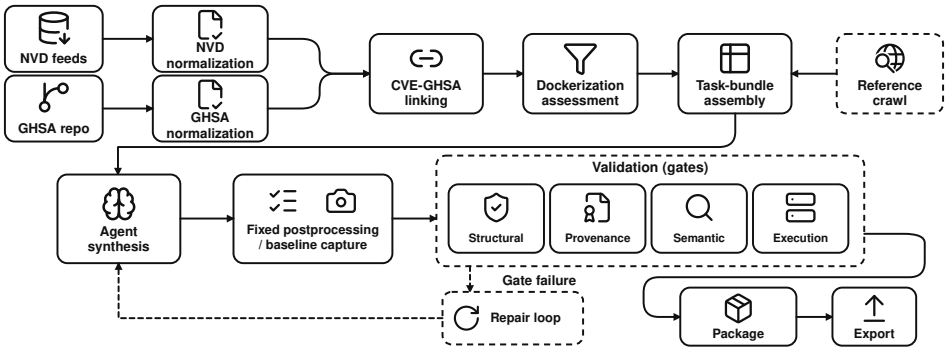

Figure 1: **Overview of the SABER construction and validation workflow.** The dashed block contains the four validation gates. Failed candidates enter the repair loop, and only candidates passing every gate are exported.

2.2 AGENT-BASED DATA SYNTHESIS

Recent work has increasingly explored using agents to automate project-specific construction of executable software-engineering data from existing repositories. SWE-smith combines LM-generated and procedural mutations with existing tests to synthesize large-scale Python tasks (Yang et al., 2025). SWE-Factory and SWE-rebench V2 use multi-agent or interactive setup workflows to infer build and test procedures (Guo et al., 2026; Badertdinov et al., 2026). Together, these systems reduce manual environment engineering while keeping tasks grounded in existing repository artifacts.

Agent-based review can further screen and refine constructed tasks. SWE-rebench V2 filters unsound instances with LLM judges (Badertdinov et al., 2026). SKT uses an agent to assess task semantics and convert failed checks into repair feedback (Tan et al., 2026).

Collectively, these studies show that agents can construct environments across heterogeneous repositories and refine task artifacts through feedback. Vulnerability-repair synthesis must recover the authentic source revision and reproduce the actual vulnerable and repaired behavior. It must also anonymize task artifacts and verify functional correctness. Accordingly, SABER grounds agent-based synthesis in pre-collected linked vulnerability evidence. Only tasks that pass structural, provenance, semantic, and execution validation are promoted.

3 METHOD

3.1 OVERVIEW

SABER transforms heterogeneous public vulnerability evidence into source-repair tasks in four stages: (1) evidence integration and preprocessing; (2) agent-based synthesis and postprocessing; (3) layered validation; and (4) feedback-driven repair. Specifically, SABER first links and deduplicates normalized NVD and GHSA records. It then assesses targets for dockerization and enriches eligible ones with crawled references. A tool-using agent constructs project-specific task artifacts, which are standardized through postprocessing. Candidates are independently validated by structural, provenance, semantic, and execution gates. Correctable failures enter feedback-driven repair, and only candidates that pass every gate are promoted. Figure 1 summarizes the SABER workflow.

3.2 EVIDENCE INTEGRATION AND PREPROCESSING

NVD provides CVE descriptions, applicability metadata, severity metrics, and external references, while GHSA provides package and version information. SABER links both sources, uses NVD metadata for assessment, and crawls external references for task synthesis. Figure 2 summarizes the pipeline and accounting for evidence integration and preprocessing.

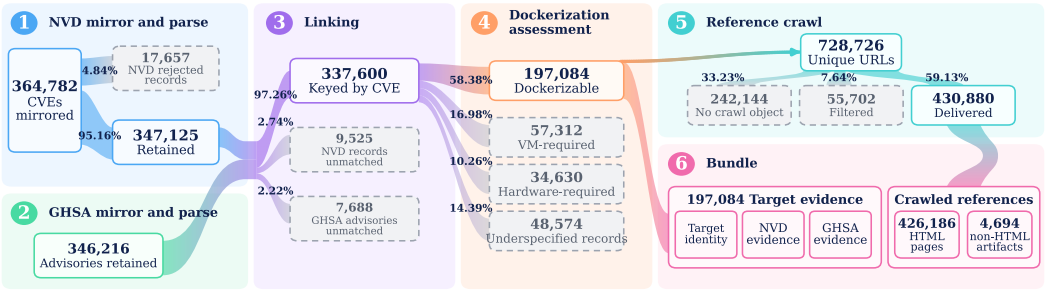

Figure 2: **Evidence integration and preprocessing.** Ribbons trace records through linking, dockerization assessment, reference crawling, and bundle construction.

NVD and GHSA normalization. We first normalize mirrored NVD 2.0 feeds and GHSA records while preserving their source-specific fields (National Institute of Standards and Technology, 2026b; GitHub, 2026). NVD entries marked `REJECTED` are removed before construction (National Institute of Standards and Technology, 2026c;a).

CVE–GHSA linking. After normalization, we link records using CVE aliases and reference URLs. Targets are then deduplicated by CVE. Matched GHSAs are aggregated into the corresponding evidence bundle, while unmatched records are excluded.

Dockerization assessment. The resulting CVE-keyed targets are screened for construction eligibility, requiring an open-source upstream repository and local executability within a Docker environment. We use applicability and platform metadata to pre-filter targets unlikely to meet these requirements. VM- or hardware-dependent targets and records with unclear applicability are discarded.

Reference crawling. We enrich retained targets by crawling advisory references and linked package pages for installation, version, patch, and PoC evidence. Inaccessible, invalid, or duplicate results are discarded. Retained HTML pages and non-HTML artifacts both enter the task bundle.

Task-bundle construction. Finally, each CVE retained after linking and screening becomes a synthesis target. SABER freezes the normalized and crawled evidence into a read-only bundle, summarized in Table 3 of Appendix A. The same bundle is used for synthesis and auditing. Among the 197,084 targets identified, 60,000 are randomly selected for subsequent synthesis.

3.3 AGENT-BASED SYNTHESIS

SABER uses a tool-using construction agent to interpret irregular or incomplete evidence and adapt synthesis to each project. We implement the agent with Codex using `gpt-5.6-sol` and `reasoning_effort=max`. Using the read-only bundle, the agent identifies the repository, vulnerable revision, and reference repair from available evidence. It then reconstructs the vulnerable environment, derives executable vulnerability checks, and produces the required task and verifier artifacts. When key provenance or faithful vulnerability reproduction cannot be established, the agent instead emits a machine-readable record of insufficient evidence.

Construction prompt and constraints. To preserve project-specific flexibility while maintaining evidential rigor, the construction prompt constrains evidence use and artifact exposure. The agent may use only supplied evidence and cited upstream history. It must preserve the official source and real vulnerable runtime. Repairs must modify production source and remain consistent with the upstream fix.

To reduce information leakage, vulnerability identifiers, exploit details, hidden verifier artifacts, and reference repairs are withheld from solver-visible content. A `compile`/`verify` interface is provided for building and PoC-based verification, while only real program observations are exposed. To accommodate alternative repairs that resolve the vulnerability, it evaluates observed PoC behavior rather than patch identity. Hidden tests further assess repair reliability for trajectory filtering. Appendix A summarizes the expected outputs and constraints in Tables 4 and 5.

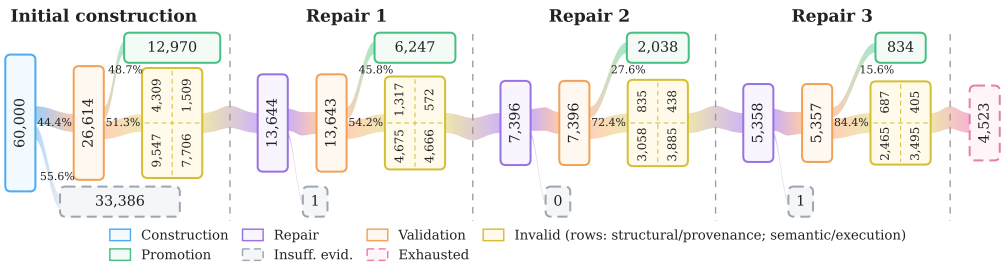

Figure 3: **Construction and repair accounting.** Ribbons trace 60,000 targets through initial construction, three repair rounds, and final budget exhaustion. Gate failures may overlap.

Postprocessing. Once the prompt-constrained agent produces a candidate, SABER normalizes it into a standard task format. It adds baseline capture scripts and supplements the verifier context with a clean vulnerable environment. The vulnerable image is built, and the PoC-based verifier runs as a non-root user in a network-isolated container. This execution captures the real baseline crash report and writes it into the corresponding task-instruction fields. The completed candidate is then sent to validation.

Together, agent synthesis and postprocessing produce a standardized task candidate. The directory structure and components of each candidate are detailed in Table 13 of Appendix C.

3.4 VALIDATION AND REFINEMENT

Flexible agent-based synthesis can introduce structural, provenance, semantic, and execution inconsistencies. SABER therefore applies four validation layers. Structured feedback from these layers is used to repair correctable failures, after which the full validation stack is rerun.

- **Structural validation.** This gate checks structural validity, metadata consistency, and patch scope. Its scope check requires a source change rather than edits confined to configuration or other non-source files.
- **Provenance validation.** An independent `gpt-5.6-sol` agent with maximum reasoning effort checks whether the repository, vulnerable state, and source snapshot are supported by the evidence. It also checks whether the repair matches the upstream fix and applies to the source.
- **Semantic validation.** Another identically configured agent checks the bug description, baseline crash report, and tests for fidelity and internal consistency. The baseline must reproduce the vulnerability and provide informative real-program output without exposing restricted details.
- **Execution validation.** An independent verifier applies the candidate patch to a clean vulnerable tree. It executes the PoC-based and hidden checks in separate vulnerable-state and reference-repair trials. Promotion requires reward records in which the vulnerable state fails and the reference repair passes, demonstrating reproduction of the vulnerable behavior and its repair, respectively.

Automatic repair. SABER converts gate failures into a structured repair prompt for correcting validation issues. These corrections must not change the target or weaken PoC-based or hidden checks. It then reruns postprocessing and the complete validation stack, recording each attempt and promoting only passing candidates. For production, a three-repair budget is used.

Promotion. Only candidates that pass the full validation stack are promoted. Digest-pinned images are prebuilt and directly used by the delivery environments.

Synthesis yield is reported for initial synthesis and each repair round. Figure 3 summarizes construction failures and repair outcomes from construction records and validation logs.

4 EXPERIMENTS

This section evaluates SABER's downstream training utility, dataset scale and diversity, construction choices, and validation controls. The experiments address five questions: (RQ1) Does fine-tuning

Table 2: **Model performance on vulnerability-repair benchmarks and SABER PHP validation (%).** Filtering denotes the selection of training trajectories.

Model	*Filtering*	**PatchEval**	**SEC-bench (Patch)**	**Vul4J**	**SABER (Val.)**
Languages		Go, JS, Python	C/C++	Java	PHP
Metric		*Strict / PoV*	*Strict / Medium / Generous*	-	*PoC+hidden/PoC*
Qwen3.5-4B	*(base)*	15.65 / 21.74	6.00 / 14.00 / 16.67	34.11	13.00 / 17.00
	None	20.00 / 20.87	20.00 / 33.67 / 34.00	39.53	20.00 / 22.00
	PoC	15.65 / 19.57	18.00 / 33.33 / 34.33	38.76	21.00 / 24.00
	PoC+hidden	**23.48 / 28.26**	**20.33 / 36.00 / 37.33**	**45.74**	**25.00 / 29.00**
Qwen3.5-35B-A3B	*(base)*	33.91 / 41.74	24.00 / 45.33 / 51.67	62.02	40.00 / 42.00
	PoC+hidden	**40.87 / 47.83**	**29.00 / 53.33 / 54.33**	**72.87**	**47.00 / 52.00**

on verified trajectories improve vulnerability-repair ability on external benchmarks? (RQ2) What are the scale and coverage of the SABER dataset? (RQ3) Which independently removable evidence-integration and preprocessing choices affect construction quality and efficiency? (RQ4) How many repair rounds are beneficial, and what does each validation gate contribute? (RQ5) How do PoC-based and hidden tests shape trajectory selection and affect downstream training?

4.1 RQ1: TRAINING UTILITY

To collect training trajectories, a teacher agent solves a language-balanced subset of 6,199 SABER tasks once each. Specifically, `deepseek-v4-flash` is run through the Claude Code harness at maximum reasoning effort. It accesses only solver-visible artifacts and runs as a non-root process in a network-isolated Docker environment. We exclude tasks sharing a CVE with the evaluation benchmarks, while allowing repository-level overlap.

We retain solutions from the 3,422 tasks that pass both the PoC-based verifier and hidden tests. Traces with unproductive loops or repetitive tool calls and tasks flagged for possible cheating are then removed. The remaining main-agent and subagent trajectories are used to fine-tune Qwen3.5-4B and Qwen3.5-35B-A3B. Table 6 in Appendix B.1 details the trajectory filtering and training configuration.

For evaluation, we compare the base and fine-tuned variants of both models on PatchEval (Wei et al., 2025), the SEC-bench patch track (Lee et al., 2025), and Vul4J (Bui et al., 2022). Table 2 reports the corresponding pass rates. For 4B and 35B-A3B, respectively, fine-tuning improves PatchEval Strict by 7.83 and 6.96 points, SEC-bench Strict by 14.33 and 5.00 points, and Vul4J by 11.63 and 10.85 points. The gains also hold under every more permissive evaluation criterion. The SABER validation set provides an additional 100-task PHP evaluation, with gains for both models.

Table 8 in Appendix B.2 further reports results stratified by repository exposure during training. The consistent gains across both groups indicate that training on filtered trajectories yields repair capabilities that generalize beyond the training repositories.

Overall, training on filtered trajectories yields improvements across model scales, benchmarks, and languages under both PoC-oriented and functionality-aware evaluation.

4.2 RQ2: DATASET SCALE AND DIVERSITY

The SABER dataset contains 22,089 tasks from 10,163 upstream repositories across more than 60 languages, with CVE-linked records spanning 1999–2026.

Figure 4 visualizes the dataset's temporal, language, and test-count profiles. 70.66% of tasks concern vulnerabilities published from 2020 onward. This concentration is consistent with increasing vulnerability discovery activity. The dataset covers most mainstream programming languages, including PHP, C/C++, JavaScript, Java, Python, and Go. PHP accounts for the largest share at 40.33%, followed by C (14.81%), JavaScript (9.37%), and Java (8.45%). Tasks contain 4–42 tests, with a median of 6. These lightweight executable checks facilitate filtering and collecting training trajectories. A human quality assessment of 50 sampled tasks is reported in Appendix B.3. Two example tasks are presented in Appendix C.

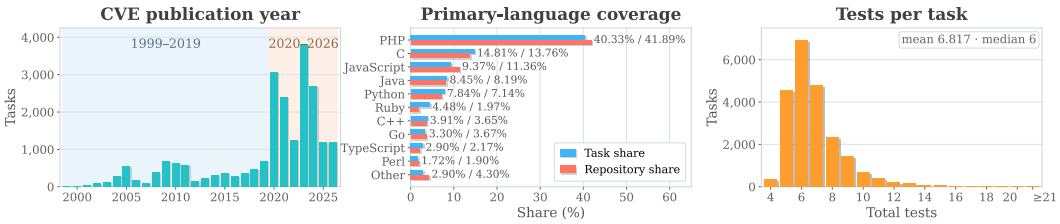

Figure 4: **SABER dataset profile:** CVE publication years (left), primary-language task and repository shares (center), and tests per task (right).

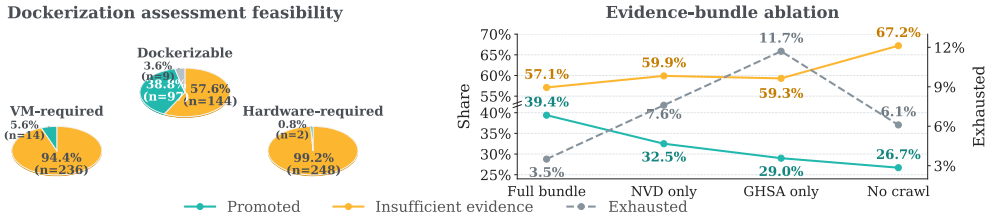

Figure 5: **Construction ablations:** dockerization assessment (left) and evidence-bundle effects (right). NVD-only and GHSA-only retain only their respective identity and evidence. No-crawl retains both identity and evidence without fetched content.

4.3 RQ3: Evidence Integration and Preprocessing Ablations

RQ3 examines two construction choices in separate controlled studies: dockerization assessment and the evidence bundle. The dockerization study samples 750 linked targets, with 250 targets in each predicted class. The evidence ablation uses a fixed 1,000-target dockerizable cohort.

We first evaluate whether deployment requirements effectively identify targets suitable for Docker-based construction. Dockerizable targets achieve a 38.8% promotion rate, compared with 5.6% and 0.8% for VM- and hardware-dependent targets. We therefore exclude the latter two classes from subsequent experiments. The left panel of Figure 5 gives the complete assessment results.

We next examine how components of the full evidence bundle affect construction efficiency. As shown in the right panel of Figure 5, the full evidence bundle lowers insufficient-evidence rates by 2.8, 2.2, and 10.1 percentage points relative to NVD-only, GHSA-only, and no-crawl, respectively. It also raises promotion rates by 6.9, 10.4, and 12.7 points. Both complementary sources and pre-crawled reference content contribute to task construction.

4.4 RQ4: Repair Budget and Validation Gates

RQ4 examines the marginal value of additional repair rounds and the independent contribution of each validation gate. Both studies use a uniform sample of 1,000 dockerizable targets. The repair-budget study allows up to five rounds, and the gate analysis aggregates synthesis and repair attempts.

The left panel of Figure 6 shows the marginal value of additional repair rounds. After excluding 576 samples owing to insufficient evidence, 424 tasks enter construction. Initial synthesis promotes 201 of these tasks, or 47.4%. By round three, 357/361 (98.9%) of eventual promotions are complete. Round four adds only four tasks and round five adds none, so we use three repair rounds in production.

The right panel of Figure 6 shows the independent contribution of each validation gate. In this sample, the four gates make complementary contributions. The provenance gate records the fewest failures but the highest unique rejection rate. Semantic and execution failures are more frequent and less often repaired. This suggests lower repairability of behavioral and execution-level defects than of structural or provenance issues.

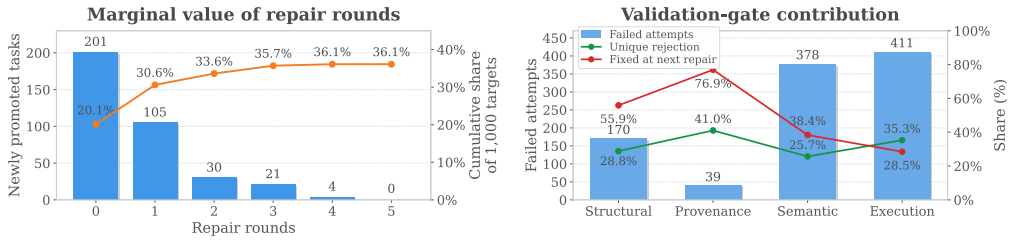

Figure 6: **Repair-budget and validation-gate analysis.** Cumulative promotion shares use all 1,000 targets. Gate-failure counts may overlap. Exact round-level counts appear in Table 11 of Appendix B.4.

4.5 RQ5: Trajectory Filtering

RQ5 examines how PoC-based verification and hidden tests affect trajectory quality and training. We analyze the same 6,199 `deepseek-v4-flash` runs used to collect the RQ1 training trajectories.

The left panel of Figure 7 quantifies filtering imposed by PoC-based and hidden tests. `deepseek-v4-flash` passes the PoC-based verifier on 4,319 tasks (69.7%) and both test layers on 3,422 tasks (55.2%). Among PoC-passing tasks, 897 (20.8%) fail at least one hidden behavior check. This suggests that some repairs satisfy the PoC-based verifier without fully restoring intended behavior.

The right panel of Figure 7 categorizes these hidden-test failures. They comprise alternative vulnerability cases, normal-operation regressions, and boundary/contrast cases. Thus, hidden tests provide complementary behavioral filtering beyond the PoC-based verifier.

Table 2 compares *None*, *PoC*, and *PoC+hidden* filtering. Appendix B.1 details the ablation settings. PoC-only filtering yields no consistent gains, while *PoC+hidden* performs best across all metrics. This indicates that hidden tests provide complementary signals for trajectory selection and training.

Runtime isolation further limits solver evasion during trajectory collection. It neutralizes 13,473 of 13,581 observed attempts (99.2%). Table 12 in Appendix B.5 gives the strategy breakdown.

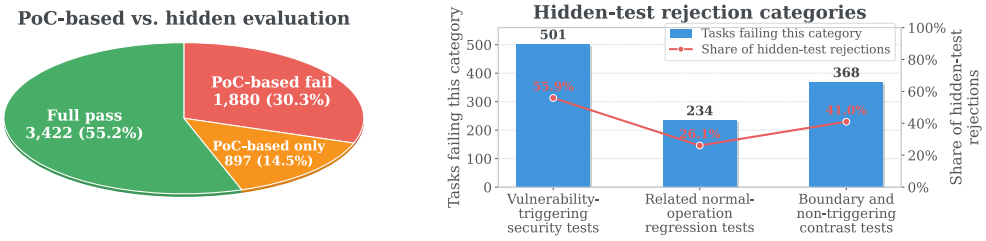

Figure 7: **PoC-based and hidden-test filtering on `deepseek-v4-flash` runs.** The right panel categorizes hidden-test failures. Categories may overlap.

5 Conclusion

We presented SABER, an agent-driven pipeline for constructing repository-level vulnerability-repair tasks from NVD, GHSA, and crawled evidence. SABER combines project-adaptive synthesis with deterministic postprocessing, four-layer validation, and feedback-driven repair. The resulting dataset contains 22,089 tasks across 10,163 repositories and more than 60 languages. Each task includes a reproducible environment, an upstream fix, and security and functionality checks. Sampled ablations demonstrate the benefit of combined evidence. Validation analysis further shows that the four gates provide complementary quality checks. Filtering ablations show that combining PoC-based and hidden tests yields higher-quality training trajectories and better model performance. Fine-tuning on filtered trajectories improves Qwen3.5-4B and Qwen3.5-35B-A3B on PatchEval, SEC-bench, and Vul4J. These results support the utility of SABER for vulnerability-repair training.

AI USE STATEMENT

In this work, we used generative AI tools to implement parts of the pipeline, assist with experimental execution, and generate executable repair-task artifacts and solver trajectories from disclosed vulnerabilities. We have not used generative AI tools for research ideation, related-work retrieval and discovery, or drafting sections of the paper. Theoretical modeling and proof-related tasks are not applicable to this work. Additionally, we used generative AI tools to aid or polish the writing of the manuscript. We have reviewed all AI-assisted work. The authors checked the text and analyses, ran AI-assisted code, screened generated tasks and trajectories using the validation and filtering procedures described in the paper, and manually audited a sample of generated tasks. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Jafar Akhoundali, Sajad Rahim Nouri, Kristian Rietveld, and Olga Gadyatskaya. MoreFixes: A large-scale dataset of CVE fix commits mined through enhanced repository discovery. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 42–51, 2024. doi: 10.1145/3663533.3664036.

Anthropic. Claude Fable 5 and Claude Mythos 5. Anthropic announcement, 2026. URL https://www.anthropic.com/news/claude-fable-5-mythos-5. Published: 2026-06-09; Accessed: 2026-08-11.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2602.23866.

Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 30–39, 2021. doi: 10.1145/3475960.3475985.

Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 464–468, 2022. doi: 10.1145/3524842.3528482.

Tan Bui, Ting Zhang, Ferdian Thung, Yunpeng Xiong, Penghao Jiang, Xin Zhou, and David Lo. Vul4Py: Benchmarking automated vulnerability repair in Python with paired exploit and functional oracles. arXiv preprint arXiv:2608.00692, 2026. URL https://arxiv.org/abs/2608.00692.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 654–668, 2023. doi: 10.1145/3607199.3607242.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1729–1741, 2025. doi: 10.1109/ICSE55347.2025.00038.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 2020. doi: 10.1145/3379597.3387501.

GitHub. GitHub advisory database. https://github.com/github/advisory-database, 2026. Accessed: 2026-08-05.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE Data Construction, Automatically! *Proceedings of the ACM on Software Engineering*, 3(FSE):525–546, 2026. doi: 10.1145/3797119. URL https://doi.org/10.1145/3797119.

Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. SoK: Automated vulnerability repair: Methods, tools, and assessments. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 4421–4440. USENIX Association, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/hu-yiwei.

Kimi Team. Kimi K3: Open frontier intelligence. arXiv preprint arXiv:2607.24653, 2026. URL https://arxiv.org/abs/2607.24653.

Amine Lbath, Manan Suri, Aurelien Delaitre, Vadim Okun, Massih-Reza Amini, Ram D. Sriram, and Dinesh Manocha. CyberForge: Verified vulnerability injection at repository level for cybersecurity agent training. arXiv preprint arXiv:2608.06471, 2026. URL https://arxiv.org/abs/2608.06471.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated benchmarking of LLM agents on real-world software security tasks. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3878. URL https://papers.nips.cc/paper_files/paper/2025/hash/a9168f1c54e5147027f1e8cf83e1a775-Abstract-Conference.html.

Xiang Mei, Jordi Del Castillo, Pulkit Singh Singaria, Haoran Xi, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, and Brendan Dolan-Gavitt. ARVO: Atlas of reproducible vulnerabilities for open-source software. In *2026 IEEE 11th European Symposium on Security and Privacy (EuroS&P)*, pp. 860–873, 2026. doi: 10.1109/EuroSP68448.2026.00060.

National Institute of Standards and Technology. CVEs and the NVD process. https://nvd.nist.gov/general/cve-process, 2026a. Accessed: 2026-08-05.

National Institute of Standards and Technology. NVD - data feeds. https://nvd.nist.gov/vuln/data-feeds, 2026b. Accessed: 2026-08-05.

National Institute of Standards and Technology. Vulnerability status - NVD. https://nvd.nist.gov/vuln/vulnerability-status, 2026c. Accessed: 2026-08-05.

OpenAI. GPT-5.6 Sol Model. OpenAI API documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol. Accessed: 2026-08-11.

Sofia Reis and Rui Abreu. A ground-truth dataset of real security patches. arXiv preprint arXiv:2110.09635, 2021. URL https://arxiv.org/abs/2110.09635.

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. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2606.04460.

Zelin Tan, Yiqun Zhang, Hao Li, Zhiyao Cui, Hejia Geng, Shao Zhang, Hangfan Zhang, Yang Chen, Xiaosong Wang, Lilong Wang, Zhenfei Yin, Shuyue Hu, Chen Zhang, and Lei Bai. SKT: Skill-use training at scale via verified synthetic data generation. arXiv preprint arXiv:2608.02287, 2026. URL https://arxiv.org/abs/2608.02287.

Zichao Wei, Jun Zeng, Ming Wen, Zeliang Yu, Kai Cheng, Yiding Zhu, Jingyi Guo, Shiqi Zhou, Le Yin, Xiaodong Su, and Zhechao Ma. PATCHEVAL: A new benchmark for evaluating LLMs on patching real-world vulnerabilities. arXiv preprint arXiv:2511.11019, 2025. URL https://arxiv.org/abs/2511.11019.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3239. URL https://papers.nips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Zheng Yu, Wenxuan Shi, Xinqian Sun, Zheyun Feng, Meng Xu, and Xinyu Xing. Patch validation in automated vulnerability repair. arXiv preprint arXiv:2603.06858, 2026. URL https://arxiv.org/abs/2603.06858.

Z.ai. GLM-5.2. Model card, 2026. URL https://huggingface.co/zai-org/GLM-5.2. Accessed: 2026-08-11.

A Construction Evidence and Prompt

Table 3 summarizes the read-only evidence supplied to the construction agent.

The tables below paraphrase the construction prompt's output requirements and behavioral constraints. Table 4 summarizes the candidate artifacts expected from the agent. Table 5 specifies the boundaries for evidence use, task authenticity, and evaluation privacy.

Table 3: Reference evidence included in the read-only construction bundle for each target.

Bundle component	**Reference content**
Target identity and linkage	Target CVE identifier and linked GHSA identifiers.
NVD evidence	Vulnerability description, weaknesses, severity metrics, CPE configuration and applicability.
GHSA evidence	Advisory text, package ecosystem and name, affected version ranges, fixed versions, source references.
Crawled references	HTML pages and non-HTML artifacts, package-manager links, and their source URLs.

Table 4: Construction-prompt requirements and expected candidate artifacts.

Prompt focus	Requirement	Expected candidate artifact
Task objective	Construct a repository-level vulnerability-repair task.	A task candidate with a vulnerable source checkout, reproducible Docker environment, PoC-based and hidden tests, and a verifier-side reference repair.
Environment construction	Rebuild and run the real vulnerable upstream revision.	A clean checkout at the evidenced vulnerable revision and a reproducible build and runtime context.
Upstream fix	Ground the reference repair in the real upstream fix.	The relevant production-source patch and a script that applies it to a clean vulnerable checkout.
PoC-based verifier	Test candidate patches on clean vulnerable source using real program behavior.	A PoC-based verifier that applies a candidate patch and checks whether the observed vulnerability behavior is eliminated without requiring patch identity.
Hidden verifier	Check additional security and functionality behavior.	A hidden matrix with vulnerable-state and reference-repair expectations for vulnerability triggers, normal operation, and boundary or contrast cases. Normal and contrast cases also assess regressions.
Task instruction and metadata	Sanitize the solver-facing instruction and record task metadata.	A sanitized `instruction.md` template with a bug description and baseline-report field, plus task metadata.
Insufficient evidence	Stop when essential evidence or faithful reproduction is unavailable.	A machine-readable insufficient-evidence record instead of a task if the repository, vulnerable revision, upstream fix, or faithful vulnerability reproduction cannot be established.

Table 5: Construction-prompt constraints on evidence, source authenticity, and solver-visible content.

Prompt focus	Requirement	Agent constraint
Evidence boundary	Use the supplied evidence and cited upstream history.	Ground the repository, vulnerable revision, and repair in those sources. Do not replace them with unsupported findings from general web search.
Source authenticity	Preserve the official vulnerable source.	Keep the vulnerable checkout as a clean baseline. Do not substitute a demo, source subset, rewritten project, or compatibility layer.
PoC authenticity	Exercise the real upstream program.	Derive the diagnostic from actual runtime behavior. Synthetic stand-ins, self-written success markers, and generic status wrappers do not establish vulnerability reproduction.
Repair scope	Require a production-source repair grounded in the upstream fix.	Version, image, configuration, test, documentation, or wrapper changes alone do not satisfy the repair requirement.
Privacy boundary	Keep restricted evidence out of solver-facing material.	Withhold vulnerability identifiers, exploit inputs, hidden tests, and the reference repair from task instructions and exposed command output.
Black-box command interface	Provide solver-facing `compile` and `verify` entry points.	Report real-program observations without exposing hidden inputs or expectations. Do not echo trigger payloads or replace behavior with generic status labels.

B Experimental Details

B.1 Training Configuration

From 6,199 `deepseek-v4-flash` runs, we select execution-verified training trajectories. Table 6 records trajectory selection and fine-tuning settings.

Table 6: Shared configuration used to fine-tune Qwen3.5-4B and Qwen3.5-35B-A3B on filtered trajectories.

Configuration	Value
Trajectory generator	`deepseek-v4-flash` (maximum reasoning effort)
Trajectory harness	Claude Code
Trajectory source	Solutions for the 3,422 full-pass tasks
Trajectory filtering	Remove 558 traces with obvious loops or repetitive tool calls and 18 tasks flagged for possible cheating
Retained trajectories	2,846 main-agent and 1,524 subagent trajectories. A task may contribute both
Trained tokens	223 million
Base models	Qwen3.5-4B and Qwen3.5-35B-A3B
Training framework	LlamaFactory
Context window	256K
Epochs	1
Global batch size	16
Optimizer steps	274
Optimizer	AdamW
Learning rate	10^{-5}
Weight decay	0.1
Learning-rate schedule	Cosine decay, 3% warmup
Seed / data seed	42 / 42

For the filtering ablations, we randomly sample 2,846 main-agent and 1,524 subagent trajectories from all collected runs (*None*) or PoC-passing runs (*PoC*). All other training settings match Table 6. Trajectory counts match the main experiment. Total trained tokens are 244 million for *None* and 225 million for *PoC*, both comparable to the main experiment's 223 million.

B.2 Training–Evaluation Overlap

We compared the 3,422 selected training instances with the evaluation benchmarks to assess potential overlap. Although repository-level overlap exists, we found no overlap in CVE identifiers or patches. Table 7 reports overlap at the repository, CVE, and patch levels. Repository-level overlap is expected, as the same project may contain distinct CVEs and fixes over time. No overlap was found at the CVE or patch level.

Table 7: Overlap between the training data and evaluation sets.

Benchmark	Total (tasks/repos)	Repository overlap (tasks/repos)	CVE	Patch
PatchEval-Verified	230/175	113/72	0	0
SEC-bench (Patch)	300/33	169/16	0	0
Vul4J	129/87	60/31	0	0
SABER (Val.)	100/100	0/0	0	0

To examine whether repository-level overlap accounts for the observed improvements, Table 8 stratifies evaluation results by whether the repository appears in the training data. Fine-tuning yields gains on both overlapping and non-overlapping subsets across the three benchmarks. In particular, consistent gains on repository-disjoint subsets indicate that the improvements extend to unseen repositories.

Table 8: Evaluation results stratified by repository overlap with the training data.

Model	Subset (n)	Base	Fine-tuned	Gain Δ
PatchEval-Verified (Strict / PoV)				
4B	Overlap (113)	19/30	28/35	+9/+5
4B	Non-overlap (117)	17/20	26/30	+9/+10
35B-A3B	Overlap (113)	36/46	47/56	+11/+10
35B-A3B	Non-overlap (117)	42/50	47/54	+5/+4
SEC-bench (Patch) (Strict / Medium / Generous)				
4B	Overlap (169)	13/24/32	39/59/60	+26/+35/+28
4B	Non-overlap (131)	5/18/18	22/49/52	+17/+31/+34
35B-A3B	Overlap (169)	43/78/92	54/91/93	+11/+13/+1
35B-A3B	Non-overlap (131)	29/58/63	33/69/70	+4/+11/+7
Vul4J (PASSING)				
4B	Overlap (60)	23	26	+3
4B	Non-overlap (69)	21	33	+12
35B-A3B	Overlap (60)	38	47	+9
35B-A3B	Non-overlap (69)	42	47	+5

B.3 Human Quality Assessment

We conducted a manual quality assessment on a language-balanced random sample of 50 SABER tasks, each from a distinct repository. Table 9 summarizes the primary-language distribution of the sampled repositories.

Table 9: Repository primary-language distribution of the 50-task assessment sample.

Language (tasks)				
PHP (5)	Java (5)	C++ (4)	Perl (4)	R (1)
C (5)	Python (5)	Go (4)	C# (1)	Rust (1)
JavaScript (5)	Ruby (5)	TypeScript (4)	Erlang (1)	
				Total: 50

Each task was evaluated along four dimensions: repository and revision validity, upstream-fix provenance and applicability, PoC-based test fidelity, and hidden-test relevance and correctness. Each dimension was assigned one of three ratings—*Pass*, *Caveat*, or *Fail*—according to the following criteria:

- **Repository/revision:** verified vulnerable repository and revision / supported with a minor provenance gap / unsupported or incorrect repository or revision.
- **Upstream fix:** source-grounded and applicable production fix / authentic but broad or downstream patch / unsupported or inapplicable fix.
- **PoC-based test:** faithful reproduction of the vulnerability with discriminative vulnerable/fixed behavior / relevant trigger with limited behavioral evidence / unrelated or non-behavioral check.
- **Hidden tests:** valid additional security and functional checks / relevant but weak or uncertain checks / incorrect expectations or acceptance of clearly ineffective repairs.

Table 10: Manual assessment results across four task-quality dimensions.

Dimension	Pass	Caveat	Fail
Repository and vulnerable revision	50	0	0
Upstream fix	43	7	0
PoC-based test fidelity	50	0	0
Regression and hidden tests	50	0	0

As shown in Table 10, 43 of the 50 tasks passed all four dimensions, while the remaining seven received only an upstream-fix caveat. No task received a failing rating. Five caveats arose from

patches containing changes beyond the vulnerability-specific repair, and two from downstream distribution security patches. Such caveats are expected when preserving source-provided fixes, which may contain broader release changes or downstream adaptations. We therefore retained the source-provided patches rather than attempting potentially error-prone manual minimization.

B.4 Repair-Budget Study

A 1,000-target dockerizable sample supports the repair-budget study. Table 11 records validation failures and promotions from initial synthesis through five repair rounds.

Table 11: Pre-production synthesis and repair statistics on 1,000 dockerizable targets. Round entrants are candidates reaching validation. Round percentages use round entrants. Cumulative percentages use all 1,000 targets. Gate-failure counts may overlap.

Repair round	**Round entrants**	**Failed validation checks, n (round %)**				**Promoted**	
		Structural	**Provenance**	**Semantic**	**Execution**	**Round**	**Cumulative**
0 (initial synthesis)	424	99 (23.3%)	26 (6.1%)	129 (30.4%)	131 (30.9%)	201 (47.4%)	201 (20.1%)
1	223	15 (6.7%)	6 (2.7%)	72 (32.3%)	88 (39.5%)	105 (47.1%)	306 (30.6%)
2	118	13 (11.0%)	4 (3.4%)	54 (45.8%)	62 (52.5%)	30 (25.4%)	336 (33.6%)
3	88	12 (13.6%)	3 (3.4%)	36 (40.9%)	39 (44.3%)	21 (23.9%)	357 (35.7%)
4	67	16 (23.9%)	0 (0.0%)	43 (64.2%)	39 (58.2%)	4 (6.0%)	361 (36.1%)
5	63	15 (23.8%)	0 (0.0%)	44 (69.8%)	52 (82.5%)	0 (0.0%)	361 (36.1%)

B.5 Evaluation Robustness

The same training solver runs also support a trajectory audit. Table 12 classifies observed evasion strategies and reports how many attempts were neutralized by runtime isolation.

Table 12: Trajectory-audit breakdown of the 13,581 observed evasion attempts. Neutralization shares use the attempt count in each row. The rate is undefined when no attempts were observed.

Evasion strategy	**Attempts**	**Neutralized by runtime isolation**
Access to protected artifacts or hidden paths	13,542	13,437 (99.2%)
Modification of a verifier, test, or build wrapper	39	36 (92.3%)
Manipulation of result metadata or rewards	0	0 (n/a)

C Representative SABER Tasks

Table 13 shows the delivered task layout. The two examples below are drawn from the published Chromium and Checkmk task packages.

Table 13: Delivered task layout.

Directory structure	Comment
harbor_task/	Task root
|-- task.toml	Task and runtime configuration
|-- instruction.md	Public task instruction
|-- environment/	Task environment
| |-- <upstream-source>/	Complete vulnerable checkout
| |-- Dockerfile	Real environment Dockerfile
| |-- compile	Build/compile or syntax-check wrapper
| `-- verify	Black-box diagnostic wrapper
|-- tests/	Verifier context
| |-- Dockerfile	Isolated verifier environment
| |-- test.sh	Verification entry point
| |-- harbor-source-original-test.sh	PoC-based verifier
| |-- harbor-source-matrix-test.sh	Hidden verifier
| `-- test_matrix.json	Hidden cases and expectations
`-- solution/	Reference solution
|-- solve.sh	Reference-solution entry point
`-- upstream_fix.patch	Upstream source fix

C.1 CHECKMK: STORED XSS IN A VIEW TITLE (PYTHON)

Project and vulnerability. Checkmk is a monitoring application. This task uses the Python GUI code in Checkmk 1.6.0p18. The vulnerability allows a stored view title containing an anchor with a `javascript:` URL to be rendered as an active link. Table 14 gives the task metadata.

Table 14: Metadata for the Checkmk task.

Field	Value
CVE	CVE-2020-28919
Upstream repository	https://github.com/tribe29/checkmk.git
Vulnerable revision	Checkmk 1.6.0p18
Primary language	Python

PoC. The verifier-side PoC passes the following value to the real `Escaper.escape_text` implementation in `cmk/gui/htmllib.py`:

```
<a href="javascript:window.__probe__">marker</a>
```

On the vulnerable revision, the observed structure is `anchor_markup=literal` with one opening anchor. After repair, it must be `anchor_markup=escaped` with zero opening anchors.

Upstream patch. The upstream repair changes the anchor matcher to capture the URL and optional `target` attribute, then parses the URL scheme before restoring an escaped anchor to literal HTML. The key logic from `solution/upstream_fix.patch` is reformatted for page width:

```
+import urlparse
...
+            href = a_href.group(1)
+            parsed = urlparse.urlparse(href)
+            if (parsed.scheme != "" and
+                    parsed.scheme not in ["http", "https"]):
+                continue
+            target = a_href.group(2)
...
+            text = text.replace(
+                a_href.group(0), unescaped_tag)
```

PoC-based test. The PoC-based verifier copies the clean source, restricts the submitted patch to `cmk/gui/htmllib.py`, applies it, and runs `python3 -m py_compile`. It then invokes `project_probe.py --case disallowed_script` against the patched `Escaper` and checks both structural observations rather than accepting an exit code alone.

Hidden tests. The nine hidden cases test the security condition and preservation of legitimate rendering. Table 15 summarizes the verifier-side matrix. "fail → pass" means the baseline violates the assertion and the repaired source satisfies it.

For instance, `disallowed_data` supplies:

```
<a href="data:text/plain,marker">marker</a>
```

This case requires zero active anchors, while `https_target` supplies an HTTPS anchor with `target="_blank"` and requires one active anchor. These cases distinguish scheme blocking from indiscriminate link removal.

Table 15: Checkmk hidden-test matrix.

Role	Hidden cases	Required observation	Baseline → fixed
Vulnerability inputs	`disallowed_script`, `disallowed_data`, `disallowed_ftp`	Unsafe-scheme anchors remain escaped. Opening-anchor count is zero.	fail → pass
Normal links	`https_link`, `relative_link`	Supported links remain literal anchors with count one.	pass → pass
Contrasts	`uppercase_http`, `https_target`, `plain_text`, `script_text`	HTTP case and target handling remain valid. Non-link text stays escaped, including the script tag entity.	pass → pass

C.2 CHROMIUM: CORS BYPASS IN THE CONTENT LOADER (C++)

Project and vulnerability. This task uses the C++ Chromium source at revision `b1df7028`, whose `chrome/VERSION` file reports version 84.0.4147.65. The bug permits a crafted page to receive cross-origin data through the Android `content://` loader when CORS policy should reject the request. Table 16 gives the task metadata.

Table 16: Metadata for the Chromium task.

Field	Value
CVE	`CVE-2020-6516`
Upstream repository	`https://chromium.googlesource.com/chromium/src.git`
Vulnerable source revision	`b1df7028`
Version in `chrome/VERSION`	84.0.4147.65
Primary language	C++

PoC. The task-level PoC is a C++ loader probe rather than the original HTML exploit: selector `3` exercises the CORS-mode request and reports the content-loader and file-loader outcomes. Its key before/after observation is:

```
$ loader_probe 3
before: loader=content response_non_cors=yes
        response=default action=deliver completion=ok
after:  loader=content response_non_cors=unknown
        response=none action=reject completion=cors-error
```

Upstream patch. The upstream patch adds response-type calculation and a rejection path to `content_url_loader_factory.cc`. It also moves the response-type helper from the file loader into shared `cors.cc`/`cors.h`, so both loaders use the same classification. The key logic of the four-file reference patch is reformatted for page width:

```
+    network::mojom::FetchResponseType response_type =
+        network::cors::CalculateResponseType(
+            request.mode, disable_web_security);
+    if ((!disable_web_security &&
+         request.mode ==
+             network::mojom::RequestMode::kSameOrigin) ||
+        response_type ==
+            network::mojom::FetchResponseType::kCors) {
+      mojo::Remote<network::mojom::URLLoaderClient>(
+          std::move(client_remote))
+          ->OnComplete(
+              network::URLLoaderCompletionStatus(
+                  network::CorsErrorStatus(
+                      network::mojom::CorsError::
+                          kCorsDisabledScheme)));
+      return;
+    }
+    head->response_type = response_type;
```

PoC-based test. The PoC-based verifier accepts patches only to the four upstream CORS and loader implementation files. It compiles those production translation units with the probe runner and runs `loader_probe 3` on the clean source to confirm the delivery diagnostic. It then applies the candidate patch, rebuilds, and requires the CORS-error rejection diagnostic. It also requires a file-loader observation. The verdict uses the loader's observed behavior.

Hidden tests. The hidden matrix runs the same compiled loader probe with five selectors. Table 17 shows the required behavior and the role of the controls.

Table 17: Chromium hidden-test matrix.

Role	Hidden cases	Required observation	Baseline → fixed
Vulnerability inputs	`cors_policy_input` (3), `same_origin_policy_input` (1), `forced_preflight_policy_input` (4)	Content requests are rejected with `cors-error`.	fail → pass
Normal request	`navigation_control` (0)	Navigation remains deliverable. The file response is `basic`.	pass → pass
Contrast	`opaque_control` (2)	No-CORS delivery remains valid. The file response is `opaque`.	pass → pass