CYBERCRAFT: SCALING AGENTIC SYNTHESIS OF REPOSITORY-LEVEL VULNERABILITY ENVIRONMENTS FOR SECURITY AGENT TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Training agents to discover and repair software vulnerabilities requires executable environments in which both a vulnerability and its remediation can be verified. However, real vulnerabilities are scarce. Synthesizing vulnerabilities is a scalable alternative, but existing methods either operate on isolated functions without an executable context or inject repository-level vulnerabilities with limited diversity. We introduce CyberCraft, an agentic framework that synthesizes diverse, verifiable repository-level vulnerability environments at scale. An injector agent, guided by an atlas of vulnerability mechanisms mined from real-world CVEs, introduces vulnerabilities into real repositories. The atlas encourages the agent to explore beyond common patterns while grounding injections in real-world mechanisms, improving diversity. The resulting environments support verified teacher demonstrations and student reinforcement learning with execution-based rewards, while the injector co-evolves with the student by targeting its failures in new environments. We construct 7,979 verified environments across 713 C/C++ projects, hold out part of them as CyberCraft-Bench, and evaluate transfer to SEC-bench, PatchEval, and CyberGym-E2E. Training on these environments substantially improves Qwen3.5-9B on the three benchmarks.

1 INTRODUCTION

Large language model (LLM) agents now carry out increasingly long software engineering tasks in large repositories (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025; Kwa et al., 2025), and the code they write can introduce vulnerabilities (Pearce et al., 2022; Perry et al., 2023; Zhao et al., 2026). Agents that can audit a repository and repair the vulnerabilities they find are therefore becoming necessary.

Training such agents requires executable environments in which a vulnerability can be triggered and its repair checked. Existing environments are mostly reconstructed from disclosed vulnerabilities (Mei et al., 2026; Lee et al., 2025; Wang et al., 2026; Luo et al., 2026), which are scarce and often hard to reproduce (Mu et al., 2018), so their supply grows only as fast as vulnerabilities are disclosed. This scarcity limits the scale and diversity of vulnerability data. Synthetic data offers a way to expand training corpora and support recursive self-improvement (Villalobos et al., 2024; Singh et al., 2024; Zhao et al., 2025; Wei et al., 2026; Good, 1965; Genewein et al., 2026). Applying this approach to open-source projects can provide a scalable source of executable vulnerability data, decoupling its growth from the rate at which vulnerabilities are discovered and disclosed.

Existing vulnerability synthesis methods fall into two broad settings. Methods that synthesize individual functions or files produce vulnerable artifacts without the repository context needed to build or execute them (Nong et al., 2023; 2024; Daneshvar et al., 2024; Lbath et al., 2025; Pushkar et al., 2025), and those that retrieve disclosed vulnerable code as a reference can transfer a poorly matched pattern to the target through similarity-based matching (Daneshvar et al., 2024; Lbath et al., 2025). Repository-level injectors preserve this context, but their guidance remains limited: some ask an LLM to introduce a bug or a specified weakness with little target-specific grounding, causing it to fall back on familiar, often trivial edits such as removing a bounds check before a memory copy; their injections consequently concentrate on a few patterns (Migliaccio et al., 2026; Wei et al., 2026).

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.

Finally, executable vulnerability pipelines commonly confirm a vulnerability through sanitizer reports (Mei et al., 2026; Wang et al., 2026; Lee et al., 2025), and synthesis that adopts this criterion is restricted to violations that those sanitizers expose.

We present CyberCraft, a framework that trains security agents on execution-verified vulnerability environments through agentic, mechanism-guided synthesis. After inspecting a target repository, the injector agent navigates a structured atlas of 1,751 vulnerability mechanisms mined from real fixes, drawing inspiration for injections suited to the code. By grounding generation in real vulnerability mechanisms and encouraging exploration beyond familiar edits, the atlas makes the environments more diverse and more useful for training. The injector harness also introduces probe-based site selection and contract-oracle verification. The probe uses execution feedback from existing inputs to identify reachable guards and guide efficient site selection. The contract oracle turns executable invariants into runtime checks, enabling verification of vulnerabilities that sanitizers miss.

We construct 7,979 verified environments from 713 C/C++ projects, covering 514 CWEs. Environments from held-out repositories form CyberCraft-Bench, to our knowledge the first repository-level, end-to-end vulnerability benchmark built through agentic LLM generation: a solver must find, reproduce, and repair each vulnerability without hints. Because its vulnerabilities are synthesized rather than collected from public disclosures, it reduces the risk of data contamination. The remaining environments train a student by supervised fine-tuning on verified teacher trajectories and then by reinforcement learning with execution-based rewards. In this stage, an environment on which every rollout fails is presented again with more information about its vulnerability, up to a triggering input, so that it still provides a learning signal. The injector co-evolves with the student: it summarizes the student's failures by CWE in a profile that it consults, along with the atlas, when generating the next round. Training with CyberCraft substantially improves Qwen3.5-9B on SEC-bench (Lee et al., 2025), PatchEval (Wei et al., 2025), and CyberGym-E2E (Shi et al., 2026), three benchmarks of real vulnerabilities.

Our contributions are as follows:

- CyberCraft, a scalable framework from vulnerability synthesis to security agent training, with an injector that co-evolves with the student and reinforcement learning that presents an unsolved environment again with more information about its vulnerability.
- A vulnerability mechanism atlas mined from real fixes and an injector harness with probe and differential contract oracle mechanisms.
- A synthetic dataset of verified environments with fine-grained metadata, the held-out CyberCraft-Bench, and gains for an open-weight LLM on three benchmarks of real vulnerabilities.

2 Related Work

Vulnerability data synthesis. Most executable security tasks are reconstructed from disclosed vulnerabilities, using OSS-Fuzz records (Mei et al., 2026; Wang et al., 2026), curated CVE collections (Wei et al., 2025), or LLM agents that rebuild environments from CVE reports, as in SEC-bench (Lee et al., 2025), CVE-Genie (Ullah et al., 2025), and CVE-Factory (Luo et al., 2026). Synthesis instead introduces vulnerabilities into existing code, at the function level or at the repository level. At the function level, synthesized vulnerable functions augment training data for vulnerability detectors. VulGen (Nong et al., 2023) and VGX (Nong et al., 2024) apply edit patterns derived from historical fixes at locations predicted by a learned model, VulScribeR (Daneshvar et al., 2024) prompts an LLM with retrieved vulnerable examples, and AVIATOR (Lbath et al., 2025) coordinates LLM agents with static analysis tools. The resulting functions cannot be built or run. Detectors trained on function-level data transfer poorly to realistic settings (Risse & Böhme, 2024; Ding et al., 2025), and whether a function is vulnerable can rarely be decided without its calling context (Risse et al., 2025). At the repository level, ZeroDayBench (Lau et al., 2026) ports 22 disclosed CVEs into similar projects by hand. GraftyVul (Ram et al., 2026) automates this porting with LLM agents, grafting findings of a commercial static analyzer into five applications and yielding 212 exploitable samples for evaluation. In CyberCraft, an injector agent decides where and how to apply mechanisms mined from fixes of more than 26,000 CVEs, an injection is accepted only with a differential witness, and the pipeline yields thousands of environments in real C/C++ projects.

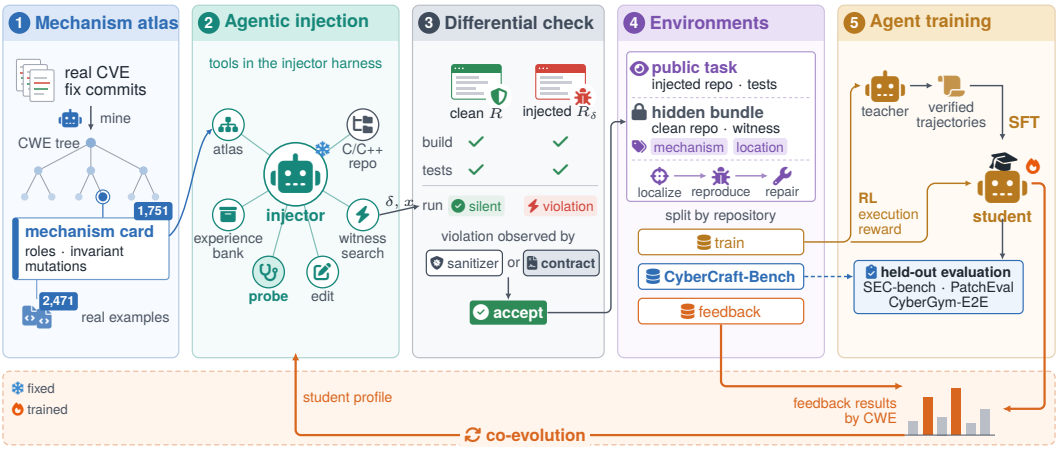

Figure 1: Overview of CyberCraft.

Security agent training. Supervised fine-tuning of security agents has used trajectories from executable CTF challenges (Zhuo et al., 2025), simulated CTF runtimes (Zhuo et al., 2026), and reconstructed CVE environments (Luo et al., 2026). Reinforcement learning has been applied to vulnerability detection with rewards from labels or LLM judges (Li et al., 2026) and to vulnerability reproduction with milestone rewards (Du et al., 2026). CyberCraft trains on synthesized environments, in which execution-based checks both select the teacher trajectories and provide the rewards for reinforcement learning. The student's failures on a feedback set also guide subsequent generation.

3 CyberCraft

3.1 Overview

CyberCraft turns open-source C/C++ projects, provisioned with reproducible builds and fuzzing entry points, into executable security environments. An environment consists of a public task and a hidden bundle. The public task contains the injected repository, its build instructions, and the tests a solver may run. The hidden bundle contains the clean revision, the witness inputs, the verification configuration, and the environment's metadata. Starting from the public task, a solver tries to localize, reproduce, and repair the vulnerability, and the hidden bundle is used to score each attempt at these three nested endpoints (Section 3.4).

Figure 1 shows how environments are produced and used. Inside the injector harness, the agent navigates the atlas (Section 3.2), runs the probe to choose a site, edits the code, and searches for a witness input. The harness accepts the injection under a differential criterion (Section 3.3) and records its metadata. Accepted environments are split by repository into a training set, a feedback set, and CyberCraft-Bench (Section 3.4). The student is fine-tuned on verified teacher trajectories from the training environments. Generation and fine-tuning run in rounds. After each round, the injector summarizes the student's results on the feedback set by CWE in a student profile, which it consults when generating the next round; its weights and the atlas stay fixed. After the last round, reinforcement learning with execution rewards continues the student's training (Section 3.5).

3.2 Vulnerability Mechanism Atlas

The atlas grounds injection in vulnerability mechanisms observed in real fixes and encourages the injector to explore beyond the edits most familiar from pretraining. It is a tool in the injector harness and a source of inspiration: after reading the target repository, the injector browses the atlas, or searches it by vulnerability type, for mechanisms and real instances that suggest injections suited to the code.

The atlas has three levels. Its skeleton is the CWE-1000 research view, whose 944 weaknesses form a tree when each keeps only the parent that MITRE marks as primary. Under each node, mechanism cards describe the distinct ways in which that weakness arises. A card is written in terms of program roles. It states the invariant among these roles that vulnerable code violates and the variants, or mutations, that break it. Under each card, examples are the real fixes from which it was derived, and through them the agent can read the relevant original code.

The atlas is populated bottom-up from the fix commits of disclosed CVEs. A DeepSeek-V4-Flash mining agent reads one CVE and its fix at a time, identifies the mechanism that the fix repairs, and either adds the fix, with its code copied verbatim, as an example of an existing card with the same roles and mutation, or creates a new card under the corresponding CWE node. Two fixes are merged only if they share the mechanism, not merely a consequence such as an out-of-bounds read, and fixes that repair no vulnerability mechanism are skipped. The sources are CVEfixes (Bhandari et al., 2021), with 12,987 fix commits for 11,726 CVEs, and the 26,961 records of the CVE list that reference a fix commit, spanning many programming languages; overlapping and repeated fixes are deduplicated. The resulting atlas contains 1,751 mechanism cards and 2,471 examples under 531 CWE nodes. By comparison, the 4,293 vulnerabilities with AddressSanitizer reports in ARVO (Mei et al., 2026) carry only 11 distinct CWE labels, and CWE-125 alone accounts for 41% of them. Appendix A.4 shows a complete card.

3.3 Agentic Injection and Differential Verification

A DeepSeek-V4-Flash injector agent works on an isolated checkout of one repository through a harness of typed tools. After reading the repository, it looks in the atlas for mechanisms whose roles bind to the code (Section 3.2).

To choose a site, the agent runs a probe, a branch census: the harness replays the existing seed corpus on a coverage build and reports, for each guard and each subcondition of a compound guard, whether seeds reach it and whether any seed takes its rejecting outcome (Appendix B.2). A guard that seeds reach but never fail is a natural site, since weakening it leaves their behavior unchanged. The census also returns the seeds that reach each guard, which start the witness search, and difficulty features such as the number of other guards on the path.

At the selected site the injector proposes a mutation and searches for a witness input through the project's fuzzing entry points. Let R be the clean snapshot, R_δ the snapshot after edit δ, and x a candidate witness. Let $B(P)$ denote a successful build of snapshot P, $T(P)$ a passing run of its regression tests, and $O_m(P, x)$ an observed violation of the injected mechanism m on input x. An attempt is accepted only if

$$A(R, \delta, x, m) = B(R) \wedge B(R_\delta) \wedge T(R) \wedge T(R_\delta) \wedge O_m(R_\delta, x) \wedge \neg O_m(R, x). \tag{1}$$

The harness evaluates this predicate on its own snapshots, which differ only in δ, with identical build flags, limits, entry point, and input; any timeout or inconclusive build counts as failure.

Contract oracles extend verification beyond what sanitizers can observe. Existing pipelines accept a vulnerability only when a sanitizer reports a crash (Mei et al., 2026; Wang et al., 2026; Lee et al., 2025), which discards violations that no sanitizer detects, such as an over-read that stays inside the same allocated object. In CyberCraft, O_m need not be a sanitizer diagnostic. When the injected mechanism admits an executable invariant, which the injector writes itself or adapts from a card, the verifier compiles it into a runtime check, the contract oracle, placed immediately before the operation that requires it, which parsing, storage, or a function boundary may separate from the weakened guard. The oracle is inserted into both snapshots, so Equation 1 also rejects an oracle that is misplaced or that the original program already violates. Contract oracles thus admit mechanisms that sanitizers leave silent (Section 4.6), and they are removed from the public task.

Across attempts on a repository, an experience bank records accepted sites and mechanisms, failed mutations with their causes, such as no witness within the budget or a failing regression test, and the inputs that reached each site. Later attempts use it to avoid used or failing sites and to seed the witness search (Appendix B).

3.4 SYNTHESIZED ENVIRONMENTS AND CYBERCRAFT-BENCH

We collect C/C++ repositories from OSS-Fuzz, ClusterFuzzLite, and Mayhem, and from the packages of Debian, vcpkg, and Conan. A repository enters the pool if, at a pinned commit, it builds with AddressSanitizer, UndefinedBehaviorSanitizer, and coverage instrumentation, has a fuzzing entry point whose seeds reach project code, and replays these seeds on its clean build without a crash. An agent writes any missing build script, entry point, or seeds, and admission depends only on these executed checks. We exclude the projects of CyberGym-E2E, SEC-bench, and PatchEval together with their forks, embedded copies, and distribution packages, and we deduplicate the pool by upstream project. We split the admitted repositories into training, feedback, and held-out groups and run the injector on each group until the group reaches its target size (Section 4.1). The injector ran on 1,572 repositories, 713 of which yielded at least one accepted environment, and the deduplicated dataset contains 7,979 environments.

The harness derives most of the metadata of each environment from the run. The mechanism and variant are recovered by matching the atlas card that the agent consulted against the shape of its diff. The CWE label is the atlas node of that card, and the observed crash class is recorded alongside it; across the dataset, the labels cover 514 distinct CWEs. The root-cause distance d is the depth of the first crash-stack frame that lies in an edited function, counting only frames in project code. An environment is local if $d = 0$, upstream if $d > 0$ or if an edited function appears only on the allocation, free, or origin stack of the sanitizer report, and off-stack otherwise. The witness type records whether a sanitizer or a contract oracle confirmed the violation. The injector also describes the vulnerability and rates each environment as easy, medium, or hard. This rating is the agent's own estimate, and Section 4.2 compares it with measured solve rates.

A solver is scored at the three endpoints of Section 3.1, each of which extends the previous one. For localization, the solver names the function that contains the vulnerability, and the answer is correct if the injection modified that function. Reproduction also requires an input that produces the designated violation in the injected build and not in the clean build, as in Equation 1. Repair further requires a patch after which the repository builds, passes its regression tests, and shows the violation on none of the witness inputs; a patch that edits the tests or stops the seeds and tests from reaching the injected site fails. The same checks select the teacher demonstrations and underlie the reward in Section 3.5.

The environments of the held-out repositories form CyberCraft-Bench, which we use to evaluate vulnerability-repair agents. It contains 379 environments, stratified by CWE and by difficulty rating. To our knowledge, it is the first repository-level, end-to-end vulnerability benchmark built through agentic LLM generation, and synthesis gives it three properties that benchmarks built from disclosed vulnerabilities lack. First, its vulnerabilities are synthesized rather than collected from public disclosures, which reduces the risk of data contamination. Second, it covers the long tail of weaknesses. Published vulnerabilities concentrate on a few CWEs: in the National Vulnerability Database (NVD), 790 of the 969 CWEs in the catalog each appear in fewer than 100 records, and together they account for only about 3% of the 312,759 records with a CWE. Benchmarks built from these records inherit this concentration, with 20 CWEs among the 200 CVE instances of SEC-bench and 41 in PatchEval-Verified. CyberCraft-Bench covers 98 CWEs, and 142 of its environments carry a CWE from the rare group, so it measures how agents handle weaknesses that are scarce in real data. Third, every environment carries the metadata described above, so results can be broken down by CWE, root-cause distance, witness type, and difficulty, and a failure can be attributed to the kind of vulnerability that caused it (Section 4.2). The benchmark evaluates an agent end to end: the agent receives only the public task, with no description of the vulnerability, its location, or a triggering input, and must localize, reproduce, and repair the vulnerability. Appendix C.5 walks through one environment.

3.5 CODE AGENT TRAINING AND CO-EVOLUTION

Teacher demonstrations and supervised fine-tuning. The teacher is a Codex agent running DeepSeek-V4-Flash. It attempts each environment from its public task together with the injector's description of the vulnerability. After the agent exits, the hidden verifier runs once, and we keep only the trajectories that pass all three endpoints. We collect three accepted trajectories for each environment, allowing at most six attempts, and replace an environment that falls short with

a new one from the same run. We fine-tune all parameters of the student, Qwen3.5-9B, on these demonstrations, each converted into a tool-use transcript that keeps the teacher's failed actions and its recovery from them (Appendix D.1).

Reinforcement learning with adaptive views. Without any information about the vulnerability, too few rollouts reach any milestone on many environments for the gradient to be informative. Each group of rollouts therefore presents its environment in one of four views, listed in increasing order of the information they take from its hidden bundle: blind, as in CyberCraft-Bench; description, which adds the description of the vulnerability; reproduction, which gives the sanitizer or oracle report of a witness input and asks only for reproduction; and repair, which also gives that input and asks only for repair. A view rewards only the milestones K_v that it leaves open, and an anti-cheat gate can set the reward to zero:

$$r(\tau) = \mathrm{Gate}(\tau) \sum_{k \in K_v} w_k \, \ell_k(\tau), \qquad K_v \subseteq \{\mathrm{loc}, \mathrm{reach}, \mathrm{poc}, \mathrm{rep}\}, \tag{2}$$

where $\ell_k(\tau) \in [0, 1]$ is the credit for milestone k and w_k its weight (Appendix D.3). A reach milestone, between the localization and reproduction checks of Section 3.4, credits a submitted input that executes the edited code in a coverage build of the injected snapshot. Each open milestone requires the one before it. The repair term ℓ_{rep} multiplies the fraction of witness inputs that no longer produce the violation by the fraction of regression tests that still pass, so passing the tests alone, which the injected program already does, earns nothing. The repair view hides the other witness inputs, so a patch that only rejects the given input earns little. The gate $\mathrm{Gate}(\tau) \in \{0, 1\}$ is zero unless the submission leaves the tests and grader untouched and the seeds and tests that reached the injected site still reach it, so deleting functionality earns nothing.

We optimize with critic-free group relative policy optimization (GRPO; Shao et al., 2024). Throughput is limited by the CPU time needed to build and fuzz environments, and a group whose rollouts all receive the same reward has zero group-relative advantage, so it spends this time without producing a gradient. Following work that filters such groups (Yu et al., 2025; Zheng et al., 2025), we first run half of each group and complete the group only if the rewards of that half differ. An environment starts in a view chosen from how often the teacher solved it. A half that fails entirely moves it to the next more informative view, and a half that succeeds entirely removes it from the pool, so an environment that the student cannot yet solve can still produce a gradient.

Co-evolving the injector with the student. Each environment records its CWE and mechanism, so the student's failures can be described in the terms the injector uses to choose what to inject, and the next round of generation can target them. In each round, the student is fine-tuned from the base model on all teacher demonstrations collected so far and attempts each environment of the feedback set (Section 3.4) once in the description view, with each attempt scored at the three endpoints. The injector then summarizes the student's performance by CWE in a short report, which forms a student profile together with the trajectories of the failed attempts. In the next round, the injector consults the profile as a reference alongside the atlas, with the report included in its instructions and the failed trajectories available through a tool; every new environment must still satisfy Equation 1.

4 Experiments

4.1 Experimental Setup

Data. CyberCraft-Bench holds 379 environments, and the feedback set holds 300, 100 for each difficulty rating. The feedback set shares no repository with the training set or CyberCraft-Bench (Section 3.4) and serves only to build the student profile: no student is trained on it, and it is not used to compare models. The remaining 7,300 environments come from two generation runs on the same 560 training repositories. The co-evolution run first generates 800 environments and then five rounds of 500, each guided by the latest student profile, for 3,300 in total. The static run generates 4,000 environments in a single pass with the same injector and atlas but without a student profile.

Training. With three verified teacher trajectories per environment, the co-evolution run has 9,900 demonstrations and the static run 12,000. Co-evolution yields six supervised checkpoints, one after

Table 1: Success rates (%) on CyberCraft-Bench, with one attempt per environment and no hints. The endpoints are nested: reproduction requires localization, and repair requires reproduction. Score is the mean of the three endpoint success rates, and the frontier models are ordered by it. Repair is also reported by the injector's difficulty rating (127 easy, 126 medium, 126 hard) and by the root-cause distance of Section 3.4 (114 local, 133 upstream, 132 off-stack). The trained rows of Qwen3.5-9B use co-evolution.

		Endpoint			**Repair by difficulty**			**Repair by distance**		
Model	**Score**	Localize	Reproduce	Repair	Easy	Medium	Hard	Local	Upstream	Off-stack
GPT-5.5	84.7	94.2	83.4	76.5	92.1	81.7	55.6	86.0	77.4	67.4
Claude Opus 4.8	79.5	90.8	77.6	70.2	88.2	75.4	46.8	80.7	70.7	60.6
Kimi K3	72.2	85.5	69.4	61.7	81.1	65.9	38.1	72.8	62.4	51.5
DeepSeek-V4-Flash	68.9	82.6	65.7	58.3	78.0	62.7	34.1	69.3	58.6	48.5
Qwen3.5-9B	14.8	24.8	11.6	7.9	15.0	6.3	2.4	14.0	6.8	3.8
+ SFT	25.6	38.8	21.4	16.6	28.3	15.1	6.3	26.3	15.8	9.1
+ SFT + RL	30.1	44.1	25.6	20.6	33.9	19.0	8.7	32.5	19.5	11.4

the initial set and one after each round. The static student is fine-tuned once on all of its demonstrations, and the two runs are compared after supervised fine-tuning. Reinforcement learning is applied to the co-evolution run only, starting from its checkpoint of the last round, for 100 steps with 32 environments per step and 8 rollouts per environment. Training uses 8 NVIDIA H200 GPUs, and Appendix D lists the hyperparameters.

Evaluation. Besides CyberCraft-Bench, we evaluate on three benchmarks of real vulnerabilities (Appendix E). SEC-bench (Lee et al., 2025) contains 200 C/C++ vulnerabilities from 29 projects, each with a proof-of-concept task and a patching task, and we report the mean of the two success rates. PatchEval (Wei et al., 2025), in its Verified release, contains 230 vulnerabilities in Go, JavaScript, and Python, languages absent from our training data. The agent receives the repository and the CVE description, and its patch must pass the vulnerability's proof-of-concept test. CyberGym-E2E (Shi et al., 2026) contains 920 vulnerabilities from 139 projects. We use its patch-only setting, in which the agent receives the source code with the ground-truth proof of concept and its crash log, and a patch succeeds if the project's tests pass and the proof of concept no longer crashes the program. The projects of these three benchmarks are excluded from the repository pool (Section 3.4), and the fixes of their vulnerabilities from the atlas. SWE-bench Verified (Jimenez et al., 2024), with 500 GitHub issues from 12 Python repositories, measures general software engineering ability. All models run in the Codex agent scaffold, and each benchmark's own checks decide success. We compare the student with Claude Opus 4.8, DeepSeek-V4-Flash, GPT-5.5, and Kimi K3 on CyberCraft-Bench, and with DeepSeek-V4-Flash, the teacher, on every benchmark.

4.2 Results on CyberCraft-Bench

CyberCraft-Bench separates agents of different strength (Table 1): the frontier models score between 68.9 and 84.7, and the base Qwen3.5-9B scores 14.8.

The difficulty rating, which the injector assigns during construction, predicts repair difficulty: every model repairs easy environments most often and hard ones least often (92.1% and 55.6% for GPT-5.5). Repair also becomes harder, again for every model, as the root cause moves farther from the crash (86.0% on local and 67.4% on off-stack environments for GPT-5.5). This ordering follows the sanitizer report: the edited function appears on the reported stack for local and upstream bugs, but for off-stack bugs it has returned before the crash, and the agent must trace the faulty value back from the crash site to where it was accepted.

The unrepaired tail is concentrated in rare CWEs (Section 3.4). For example, no frontier model repairs any environment under CWE-468 (incorrect pointer scaling) or CWE-806 (a destination buffer accessed with the size of the source buffer). NVD records only 4 vulnerabilities under CWE-468 and none under CWE-806, so training data selected by CWE label contains few such cases. Because the atlas classifies each fix by its code, synthesis can still supply training environments for these weaknesses.

Table 2: Success rates (%) of Qwen3.5-9B students trained on the static dataset (4,000 environments, supervised fine-tuning only) or with co-evolution (3,300 environments), both generated from the same training repositories. The Teacher column gives the model that produced the demonstrations (Section 4.5), and w/o atlas removes the atlas from the injector (Section 4.6). Fixed denotes reinforcement learning with the description view for every rollout group instead of adaptive views (Section 3.5). Small gray numbers give the change from the base model; bold marks the best student. CyberCraft-Bench reports the score of Table 1, with one attempt per environment and no hints. The SEC-bench score averages its proof-of-concept and patching success rates.

		Synthetic	**Real vulnerabilities**			**General**
Model	**Teacher**	CyberCraft-Bench	SEC-bench	PatchEval	CyberGym-E2E	SWE-bench V.
Reference models						
DeepSeek-V4-Flash	–	68.9	67.0	80.4	77.7	73.6
Qwen3.6-27B	–	20.6	31.0	24.3	19.5	68.8
Qwen3.5-9B	–	14.8	12.0	15.2	11.4	41.6
Static dataset						
+ SFT	V4-Flash	21.1+6.3	25.5+13.5	27.4+12.2	18.5+7.1	42.4+0.8
+ SFT	27B	18.7+3.9	21.5+9.5	19.1+3.9	15.9+4.5	42.0+0.4
+ SFT, w/o atlas	27B	15.3+0.5	15.0+3.0	16.5+1.3	13.2+1.8	41.8+0.2
Co-evolution						
+ SFT	V4-Flash	25.6+10.8	31.5+19.5	31.7+16.5	21.7+10.3	42.2+0.6
+ SFT + RL, fixed	V4-Flash	26.9+12.1	33.0+21.0	35.7+20.5	22.9+11.5	42.4+0.8
+ SFT + RL	V4-Flash	**30.1**+15.3	**35.5**+23.5	**37.0**+21.8	**24.8**+13.4	**42.8**+1.2

4.3 Transfer to Real Vulnerabilities

Training only on synthesized vulnerabilities improves Qwen3.5-9B on all three benchmarks of real vulnerabilities, with either dataset (Table 2). With co-evolution, supervised fine-tuning followed by reinforcement learning improves the student by 13.4 to 23.5 points on these benchmarks, and reinforcement learning adds to the gain of supervised fine-tuning on each of them. The gain on PatchEval (21.8), whose vulnerabilities are all in Go, JavaScript, or Python, is close to that on SEC-bench (23.5), whose vulnerabilities are in C/C++, so what the student learns from C/C++ environments carries over to other languages. Averaged over the three benchmarks, the trained student closes 31% of the gap between the base model and DeepSeek-V4-Flash, its teacher. Its score on SWE-bench Verified rises by 1.2 points, indicating that training on security environments preserves general software engineering ability.

To isolate the adaptive views of Section 3.5, we repeat reinforcement learning from the same checkpoint with every rollout group in the description view. The resulting student is weaker than the adaptive-view student on all four security benchmarks, and its average gain over supervised fine-tuning is 2.0 points instead of 4.2.

4.4 Effect of Co-evolution

After supervised fine-tuning, the co-evolved student outperforms the static one on all four security benchmarks (Table 2), although it is trained on 700 fewer environments and 2,100 fewer demonstrations. Its gain over the base model is 1.4 to 1.7 times that of the static student on each benchmark.

CWE-806 shows how the injector responds to a weakness named in a profile and what the student gains. After round 2, the student repaired none of the six CWE-806 environments in the feedback set. According to the profile, it found the buffer copy but not the weakened check, and it stopped fuzzing when no crash appeared, because the overflow stayed inside an object (Appendix D.4). In rounds 3 to 5, 95 of the 1,500 environments that the injector generated fall under CWE-806; in the static run, 80 of 4,000 do. Of the seven CWE-806 environments in CyberCraft-Bench, none of which a frontier model repairs (Section 4.2), the co-evolved student reproduces three and repairs two, and the static student reproduces one and repairs none.

4.5 Effect of a Smaller Teacher

To test whether CyberCraft depends on a strong teacher, we repeat supervised fine-tuning on the static dataset with Qwen3.6-27B in place of DeepSeek-V4-Flash as the teacher, collecting demonstrations in the same way. Although Qwen3.6-27B is a weak security agent, averaging 23.9 over the four security benchmarks, its student still improves over the base model on all four (Table 2), by 5.5 points on average, closing 52% of the gap to its teacher (16% with DeepSeek-V4-Flash). Because only trajectories that pass the verifier are kept, collection is rejection sampling from the teacher (Yuan et al., 2023): the student imitates the teacher's verified successes over up to six attempts, and a weak teacher costs attempts and replacement environments but adds no wrong demonstrations.

4.6 Effect of the Atlas and the Contract Oracle

To isolate the atlas, we build a second static dataset with the atlas tools removed from the injector harness, keeping the injector, probe, acceptance criterion, and the 560 training repositories of the static run unchanged, until it also holds 4,000 environments. Without the atlas, the injector explores freely and must invent both the mechanism and the site, and its environments collapse onto a few familiar weaknesses. Labeled by the CWE of the card that best matches each diff, they cover 68 CWEs instead of 407, and the five most frequent CWEs account for 43.4% of them instead of 15.2%. The injector falls back on edits familiar from pretraining, such as removing a bounds check before a memory write, and rarely reaches mechanisms such as a copy bounded by the source length (CWE-806) or an unvalidated quantity used as a length (CWE-1284). We fine-tune Qwen3.5-9B on this dataset with Qwen3.6-27B as the teacher and compare it with the student trained on the static dataset with the same teacher (Section 4.5). Without the atlas, the student is weaker on all four security benchmarks (Table 2), and its average gain over the base model falls from 5.5 to 1.7 points. The diversity that the atlas adds therefore carries over to the student's performance.

To measure what the contract oracle adds, we replay the witness of every environment under a criterion that accepts only sanitizer reports. Of the 7,979 environments, 1,100 (13.8%) fail it because no sanitizer reports their violation, so from the same injection attempts the oracle yields 16% more environments than a sanitizer-only criterion. These oracles do not fire on the original code: fuzzing the clean snapshot of each of the 1,100 environments with its oracle compiled in found no violation. The oracle also admits weaknesses in forms that no sanitizer sees, such as a copy that overflows an array field into the next field of the same structure (CWE-806; Appendix C.6) or an unsigned wraparound that bypasses a length check while the access stays inside an allocation (CWE-190). Because the public task contains no oracle, a solver that triggers such a violation sees no crash and must construct the witness from the code, which is where the student failed on CWE-806 (Section 4.4).

5 Conclusion

We introduced CyberCraft, a framework that synthesizes repository-level vulnerability environments for training security agents. Its injector agent draws inspiration from an atlas of 1,751 vulnerability mechanisms mined from the fixes of more than 26,000 CVEs. From 713 C/C++ projects, we built 7,979 execution-verified environments covering 514 CWEs, including the held-out CyberCraft-Bench. We trained a Qwen3.5-9B student by supervised fine-tuning and agentic reinforcement learning with execution-based rewards, while the injector co-evolved with it, generating new environments that target its weaknesses. The student improves substantially on three benchmarks of real vulnerabilities, showing that skills learned from synthesized vulnerabilities transfer to real ones. Co-evolution also yields a stronger student than a larger static dataset. CyberCraft thus offers the community a scalable way to address the scarcity of training environments for security agents.

AI Use Statement

LLM agents are part of our method: they mine the atlas, prepare repositories, inject the vulnerabilities, and produce the teacher demonstrations (Section 3), and every environment they produce is verified by execution (Equation 1). We also used LLMs to write part of the code and to edit the

manuscript. The authors checked all AI-assisted code and text and take full responsibility for this work.

Ethics Statement

This work synthesizes vulnerabilities in order to train agents that find and repair them, so it is dual-use. All injection and verification runs on isolated copies of open-source repositories. We submit nothing upstream and disclose no new vulnerability in real software. We will release the vulnerable code and triggering inputs only for research use.

Reproducibility Statement

Section 3 describes the method, Section 4.1 the experimental setup, and Appendices F, B, and D give the prompts, tool specifications, and hyperparameters. We will release the code, the atlas, the environments, and CyberCraft-Bench.

References

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 (PROMISE)*, pp. 30–39, 2021.

Seyed Shayan Daneshvar, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. VulScribeR: Exploring RAG-based vulnerability augmentation with LLMs. *arXiv preprint arXiv:2408.04125*, 2024.

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 *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1729–1741, 2025. arXiv:2403.18624.

Mingzhe Du, Luu Anh Tuan, Tianyi Wu, Renyang Liu, Zhijiang Guo, Dong Huang, and See-Kiong Ng. Mastermind: Strategy-grounded learning for repository-scale vulnerability reproduction. *arXiv preprint arXiv:2607.01764*, 2026.

Tim Genewein, Matija Franklin, Alexander Lerchner, Laurent Orseau, Samuel Albanie, Adam Bales, Cole Wyeth, Stephanie Chan, Iason Gabriel, Joel Z. Leibo, Allan Dafoe, Marcus Hutter, Thore Graepel, and Shane Legg. From AGI to ASI. *arXiv preprint arXiv:2606.12683*, 2026.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in Computers*, volume 6, pp. 31–88. Academic Press, 1965. doi: 10.1016/S0065-2458(08)60418-0.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Chris Painter, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2503.14499.

Nancy Lau, Louis Sloot, Jyoutir Raj, Giuseppe Marco Boscardin, Evan Harris, Dylan Bowman, Mario Brajkovski, Jaideep Chawla, and Dan Zhao. ZeroDayBench: Evaluating LLM agents on unseen zero-day vulnerabilities for cyberdefense. In *ICLR Workshop on Agents in the Wild*, 2026. arXiv:2603.02297.

Amine Lbath, Massih-Reza Amini, Aurelien Delaitre, and Vadim Okun. AVIATOR: Towards AI-Agentic vulnerability injection workflow for high-fidelity, large-scale code security dataset. *arXiv preprint arXiv:2508.20866*, 2025.

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 (NeurIPS)*, 2025. arXiv:2506.11791.

Youpeng Li, Fuxun Yu, and Xinda Wang. From SFT to RL: Demystifying the post-training pipeline for LLM-based vulnerability detection. *arXiv preprint arXiv:2602.14012*, 2026.

Xianzhen Luo, Jingyuan Zhang, Shiqi Zhou, Jinyang Huang, Chuan Xiao, Qingfu Zhu, Zhiyuan Ma, Xing Yue, Yang Yue, Wencong Zeng, and Wanxiang Che. CVE-Factory: Scaling expert-level agentic tasks for code security vulnerability. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.03012.

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 *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 860–873, 2026. arXiv:2408.02153.

Luca Migliaccio, Roberto Natella, Naghmeh Ivaki, Nuno Laranjeiro, and Marco Vieira. Automated vulnerability injection in smart contracts using large language models. *arXiv preprint arXiv:2609.02624*, 2026.

Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *USENIX Security Symposium*, 2018.

Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. VULGEN: Realistic vulnerability generation via pattern mining and deep learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 2527–2539, 2023.

Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.

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 *IEEE Symposium on Security and Privacy (S&P)*, 2022. arXiv:2108.09293.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023. arXiv:2211.03622.

Chinmay Pushkar, Sanchit Kabra, Dhruv Kumar, and Jagat Sesh Challa. Beyond single bugs: Benchmarking large language models for multi-vulnerability detection. *arXiv preprint arXiv:2512.22306*, 2025.

Omri Ram, Mitchell Horner, Ron van der Meyden, Alsharif Abuadbba, and Hammond Pearce. GraftyVul: Synthesising insecure programs through real-world vulnerability grafting. *arXiv preprint arXiv:2608.27928*, 2026.

Niklas Risse and Marcel Böhme. Uncovering the limits of machine learning for automatic vulnerability detection. In *USENIX Security Symposium*, 2024. arXiv:2306.17193.

Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):388–410, 2025. arXiv:2408.12986.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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 *International Conference on Machine Learning (ICML)*, 2026. arXiv:2606.04460.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *Transactions on Machine Learning Research (TMLR)*, 2024. arXiv:2312.06585.

Saad Ullah, Praneeth Balasubramanian, Wenbo Guo, Amanda Burnett, Hammond Pearce, Christopher Kruegel, Giovanni Vigna, and Gianluca Stringhini. From CVE entries to verifiable exploits: An automated multi-agent framework for reproducing CVEs. *arXiv preprint arXiv:2509.01835*, 2025.

Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Position: Will we run out of data? Limits of LLM scaling based on human-generated data. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2211.04325.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2407.16741.

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 (ICLR)*, 2026. arXiv:2506.02548.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2512.18552.

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.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.15793.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2503.14476.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute Zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2505.03335.

Songwen Zhao, Danqing Wang, Kexun Zhang, Jiaxuan Luo, Zhuo Li, and Lei Li. Is vibe coding safe? Benchmarking vulnerability of agent-generated code in real-world tasks. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2512.03262.

Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2506.02177.

Terry Yue Zhuo, Dingmin Wang, Hantian Ding, Varun Kumar, and Zijian Wang. Training language model agents to find vulnerabilities with CTF-Dojo. *arXiv preprint arXiv:2508.18370*, 2025.

Terry Yue Zhuo, Dingmin Wang, Hantian Ding, Varun Kumar, and Zijian Wang. Cyber-Zero: Training cybersecurity agents without runtime. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2508.00910.

A VULNERABILITY MECHANISM ATLAS

A.1 SKELETON

The skeleton is the CWE-1000 research view at version 4.20. Of its 944 weaknesses, 144 have more than one parent. Keeping for each weakness only the parent that MITRE marks as primary turns the view into a tree with the 10 Pillars as roots and a depth of at most five. A card may be attached only to a node whose MITRE mapping guidance is Allowed or Allowed-with-Review, which holds for 842 nodes. The other 102 nodes, 58 marked Prohibited and 44 Discouraged, include broad classes such as CWE-119 and CWE-20. They remain in the tree for navigation, but a fix labeled with one of them is filed under a more specific descendant. The injector receives the whole tree in a single call, since its rendering takes about 15k tokens.

A.2 CARDS AND EXAMPLES

Table 3 lists the fields of a card. The invariant is the only field written as a predicate over values, with the roles of the card as its identifiers, so that the injector can reuse it for the contract oracle; like the rest of the card it is a reference, and the injector may write its own invariant instead, for a mechanism with or without a card. A mechanism that concerns the order of events, such as a use after free, has no honest value predicate. Its card carries no invariant, and such injections are accepted on a differential sanitizer report alone.

Table 3: Fields of a mechanism card. The provenance is kept for auditing and is not shown to the injector.

Field	Content
Name	One line of at most 200 characters, shown in the tree and in search results.
Summary	The mechanism in prose: what correct code maintains, how the protection is lost, and what the failure becomes.
Recognize by	At least two conditions visible in the correct code, which let the injector find sites at which the invariant currently holds.
Roles	The program elements that the mechanism involves. The invariant is written over them, and the injector binds each to an expression in the target.
Invariant	A predicate over the roles that correct code maintains, compiled into the contract oracle. Omitted for ordering mechanisms.
Variants	The shapes in which the mutation occurs: for each, the code before and after it, and constraints under which the mutated code still reads as code a maintainer could have written.
Observable	What the violation looks like when it occurs, including when a sanitizer reports it.
CWE node	The single node to which the card is attached, one of the 842 attachable nodes.
Provenance	The fix commits from which the card was derived.

An example is one hunk of a real fix, shown on both sides: the fixed code, which is ordinary correct code, and the code before the fix, which shows the mechanism. Each side is a verbatim excerpt of the commit with its surrounding context, and anchors mark the line at which each role occurs. An anchor must match its excerpt exactly once, so no line written by the mining agent can enter an example. The fixed side comes with the card when the injector opens it; the vulnerable side is fetched separately, with a budget of 3 excerpts per attempt, so that the injector studies the mechanism rather than copying a known edit.

A.3 MINING

The mining agent processes one fix at a time. It receives the CVE record, the fix commit with its diff and surrounding context, and read access to the current atlas through the same browse and search tools that the injector uses. It returns one of three decisions: the fix is an instance of an existing card, and it becomes an example with role anchors unless it repeats the change of a recorded example; the fix shows a mechanism that no card describes, and a new card is created under a node that the agent proposes; or the fix does not repair a vulnerability mechanism, as with version bumps, changes to tests only, or refactorings, and it is skipped. Before mining, fixes are deduplicated by commit, both

across the two sources and across CVEs that share a fix. Two fixes share a card only if their roles and mutation are the same. The node proposed by the agent must be attachable and is otherwise pushed down to a descendant, and the CWE label of the source record is used only as a hint, since 15% of such labels name no weakness of the view. A decision is committed only after the checks of Table 3 and the anchor check pass. The agent processed every fix from the two sources of Section 3.2: the 12,987 fix commits of CVEfixes and the 26,961 CVE records that reference a fix commit. Of these, 2,471 were recorded as examples, 1,751 of them creating a new card and the rest extending one; the others were removed as duplicates, repeated the change of a recorded example, or did not repair a vulnerability mechanism. The 1,751 cards cover 531 of the 842 attachable nodes.

A.4 Example Mechanism Card

Figure 2 shows a card as the injector reads it, and Figure 3 shows three of its ten examples. The card is attached to CWE-806 (Buffer Access Using Size of Source Buffer), a Variant node on the path CWE-664, CWE-118, CWE-119, CWE-805, CWE-806 of the skeleton. The MITRE entry for CWE-806 cites no CVE as an observed example, and NVD files no record under it (Section 4.2), yet every example of the card is the fix of a disclosed CVE: five in the Linux kernel and one each in BlueZ, OpenSC, Net-SNMP, file, and memcached.

In all ten fixes the copy was bounded by a quantity taken from the source, and the fix bounded it by the destination. The fixes therefore share the roles and the mutation and form one card, but the vulnerable code takes four shapes, which the card records as variants. In E1 and E4 the length was checked only against the input that holds the item (V1). In E2 the size argument of the copy was the length of the source string (V2). In E7 the clamp to the destination stood after the copy that it should have bounded (V3), and in the remaining fixes the copy used the length that the source declared (V4). To use the card, the injector binds each role to an expression in the target repository, and under this binding the verifier compiles the invariant into the contract oracle (Appendix B.3). The environment in Appendix C.5 applies variant V1.

The examples also show why this mechanism needs the contract oracle. In E1, E2, E4, and E9, and in the first copy of E3, the destination is an array field followed by other fields of the same object, so a short overflow stays inside one allocation, where a sanitizer places no redzone and reports nothing. In E3 the two copies differ only in the field that they write: an overflow of `primary` runs into `secondary`, whereas an overflow of `secondary`, the last field of the structure, leaves the object and is reported. The oracle compiled from the invariant reports both.

Mechanism card `mech.copy.bounded_by_source_length.v1`

Name. Copy into a fixed-capacity destination bounded by the length of the source.

Summary. Code copies an item that arrives from outside the program, such as a record field, a protocol parameter, a string, or a caller's buffer, into a destination whose capacity was fixed when it was declared: an array field of a structure, a local array, or a buffer allocated with a constant size. Correct code bounds the number of elements written by that capacity, with a clamp or with a check that rejects the item. In the vulnerable code the only bound on the copy comes from the source: the item's own length field, the number of input bytes that remain, or the length of the source string. Every input the authors had in mind satisfies both bounds, so the code works until an item longer than the destination arrives, and the copy then writes past the destination. When the destination is an array field, a short overflow overwrites the fields that follow it in the same object.

Recognize by. (1) A destination whose capacity is a compile-time constant: an array field of a structure, a local array, or a buffer allocated once with a fixed size. (2) A source item whose length is taken from the input: a length byte or field of a record, a type-length-value element, a size supplied by the caller, or the length of a received string. (3) A bound on the copy that names the destination, such as `sizeof(dst)`, a capacity constant, a clamp like `MIN(len, sizeof(dst))`, or a comparison that rejects the item, often next to a second test that keeps the item inside the input.

Roles. `dst`: the destination buffer that the copy writes into. `dst_cap`: the capacity of `dst`, in the units of the copy. `src`: the source item, located in the input. `src_avail`: how many elements the input provides for the item: its declared length or the number of bytes that remain. `len`: the number of elements the copy is asked to transfer. `written`: the number of elements the copy writes into `dst`, counting a terminator it appends. `copy`: the operation that writes into `dst`, such as `memcpy`, `strncpy`, `snprintf`, `copy_from_user`, or a loop; the invariant must hold just before it.

Invariant. `written <= dst_cap`, evaluated immediately before `copy`.

Variant V1. Bound the item by what the source holds instead of by the capacity of the destination (E1, E4).

```
if (len > dst_cap || off + len > src_end) goto reject;        correct
copy(dst, src + off, len);
if (off + len > src_end) goto reject;                         mutated
copy(dst, src + off, len);
```

Constraints: only where the input can supply an item longer than `dst_cap`, so that the capacity term is the only thing keeping `len` below it; keep the source test and its error path, so the item is still validated and the edit reads as a check that covers one of its two limits.

Variant V2. Take the size argument of a bounded copy from the source (E2).

```
strlcpy(dst, src, sizeof(dst));                               correct
strncpy(dst, src, strlen(src) + 1);                           mutated
```

Constraints: only where `src` can be longer than `dst` on a reachable path, for example a name or key taken from a request; the edit must read as copying the whole string with its terminator, which is the usual reason to write `strlen(src) + 1`.

Variant V3. Apply the clamp to the destination capacity after the copy instead of before it (E7).

```
len = min(len, sizeof(dst));                                  correct
copy(&dst, src, len);  ...  copy_out(out, &dst, len);
copy(&dst, src, len);  ...                                    mutated
len = min(len, sizeof(dst));  copy_out(out, &dst, len);
```

Constraints: only where a later copy out of `dst` also needs the clamp, so that the clamp still has a use where it now stands; keep the clamp and its comment unchanged; only its position moves.

Variant V4. Drop the clamp to the destination capacity and copy the length the source declares (E3, E5, E6, E8, E9, E10).

```
n = MIN(len, sizeof(dst));                                    correct
memcpy(dst, src, n);
memcpy(dst, src, len);                                        mutated
```

Constraints: only where `len` comes from the input rather than from a constant; keep any test that the input holds `len` bytes, so that only the destination side of the copy overflows.

Observable. An out-of-bounds write of `written - dst_cap` elements at `copy`. AddressSanitizer reports it only when the write reaches memory outside the enclosing object. When `dst` is an array field followed by other fields of the same object, a short overflow stays inside the allocation and no sanitizer reports it; when `dst` is a local array or the last field of its object, the write reaches a redzone and is reported. The contract oracle compiled from the invariant detects both cases.

CWE node. CWE-806, Buffer Access Using Size of Source Buffer.

Provenance (not shown to the injector): E1 CVE-2013-1929, Linux tg3 driver, V1; E2 CVE-2013-2850, Linux iSCSI target, V2; E3 CVE-2023-50229/50230, BlueZ PBAP client, V4; E4 CVE-2018-16392, OpenSC TCOS driver, V1; E5 CVE-2022-24805, Net-SNMP VACM table, V4; E6 CVE-2019-8906, file, ELF core notes, V4; E7 CVE-2013-1828, Linux SCTP socket option, V3; E8 CVE-2013-4512, Linux UML exit code, V4; E9 CVE-2013-4514, Linux wlags49_h2 driver, V4; E10 CVE-2020-10931, memcached binary protocol, V4.

Figure 2: A mechanism card of the atlas under CWE-806, in the form in which the injector reads it. The correct and mutated shapes of each variant are written in terms of the roles, and each variant lists the examples that show it. Figure 3 shows three of the ten examples.

E1 CVE-2013-1929 Linux tg3 driver, `tg3_read_vpd()`, commit `715230a4` V1

```
          len = pci_vpd_info_field_size(&vpd_data[j]);                    len
          j += PCI_VPD_INFO_FLD_HDR_SIZE;
          if (j + len > block_end)                                        src_avail
              goto partno;
-         memcpy(tp->fw_ver, &vpd_data[j], len);          bounded only by the VPD block
-         strncat(tp->fw_ver, " bc ", vpdlen - len - 1);
+         if (len >= sizeof(tp->fw_ver))                                  dst_cap
+             len = sizeof(tp->fw_ver) - 1;
+         memset(tp->fw_ver, 0, sizeof(tp->fw_ver));
+         snprintf(tp->fw_ver, sizeof(tp->fw_ver), "%.*s bc ", len,       copy, dst
+             &vpd_data[j]);
      }
  partno:
```

```
  ...
      len = pci_vpd_info_field_size(&vpd_data[i]);
      i += PCI_VPD_INFO_FLD_HDR_SIZE;
      if (len > TG3_BPN_SIZE ||                                    part number: capacity and block
        (len + i) > vpdlen)
          goto out_not_found;
      memcpy(tp->board_part_number, &vpd_data[i], len);
```

The firmware version read from the device's VPD is copied into `fw_ver` with a bound taken only from the VPD block, while the part number below is bounded by both its capacity and the block. `fw_ver` is a 32-byte field of `struct tg3`, followed by counters and pointers.

E2 CVE-2013-2850 Linux iSCSI target, commit `cea4dcfd` V2

```
  struct iscsi_extra_response {
-     char key[64];
+     char key[KEY_MAXLEN];                                          dst, dst_cap
      char value[32];                                               follows dst in the same object
      struct list_head er_list;
  } ____cacheline_aligned;
  static int iscsi_add_notunderstood_response(
  ...
      if (strlen(value) > VALUE_MAXLEN) {                           checks value, which is not copied
  ...
      extra_response = kzalloc(sizeof(struct iscsi_extra_response), GFP_KERNEL);
  ...
-     strncpy(extra_response->key, key, strlen(key) + 1);           bound taken from the source
-     strncpy(extra_response->value, NOTUNDERSTOOD,
-             strlen(NOTUNDERSTOOD) + 1);
+     strlcpy(extra_response->key, key, sizeof(extra_response->key));   copy, dst, dst_cap, src
+     strlcpy(extra_response->value, NOTUNDERSTOOD,
+         sizeof(extra_response->value));
```

The key comes from the initiator's login request. `iscsi_check_key()` treats a key longer than `KEY_MAXLEN` as not understood, and its caller passes exactly such a key to this function, whose copy is bounded by the key itself.

E3 CVE-2023-50229/50230 BlueZ PBAP client, `read_version()`, commit `5ab53525` V4

```
  struct pbap_data {
  ...
      uint8_t primary[16];                                          dst of the first copy
      uint8_t secondary[16];                                        dst of the second copy, last field
  };
  if (!g_obex_apparam_get_bytes(apparam, PRIMARY_COUNTER_TAG, &data,   src
                                &len)) {                                len
  ...
- if (memcmp(pbap->primary, data, len)) {
+ if (len == sizeof(pbap->primary) && memcmp(pbap->primary, data, len)) {   dst_cap
      memcpy(pbap->primary, data, len);                                 copy, dst
  ...
- if (memcmp(pbap->secondary, data, len)) {
+ if (len == sizeof(pbap->secondary) &&
+         memcmp(pbap->secondary, data, len)) {
      memcpy(pbap->secondary, data, len);
```

The counters come from the phone-book server. An overflow of `primary` by up to 16 bytes lands in `secondary`, inside the same heap object, and AddressSanitizer does not report it; any overflow of `secondary`, the last field, leaves the object and is reported. The contract oracle reports both.

Figure 3: Three examples of the card in Figure 2, quoted from their fix commits. Red lines are removed by the fix and green lines added. Blue tags give the role that a line binds in the example's record, and gray tags are our annotations. Dots mark elided lines, and blank lines are omitted.

B INJECTOR HARNESS

B.1 TOOLS

The injector works inside a container without network access and has no shell. Table 4 lists its tools. Every tool that builds, runs, or measures returns an evidence identifier and writes a receipt to a ledger that the agent cannot modify, and the harness reads the metadata of an environment from this ledger rather than from the agent's messages. Each build, run, and fuzzing job executes in a container with no network access, 2 CPU cores, 4 GB of memory, and a cap of 512 processes, so a build or an input can neither reach the network nor exhaust the host. The verifier owns the snapshots and the configuration: `verify.run` rebuilds both trees from content-addressed snapshots and re-evaluates Equation 1 itself, so an environment is certified by the harness rather than by anything the agent reports. A repository whose build fails 5 times in a row is set aside for the rest of the run.

Table 4: Tools of the injector harness.

Tool	Input and output
Atlas	
`atlas.map`	The skeleton as an outline, with the number of cards under each node.
`atlas.browse`	A node; its children and the names and recognition conditions of its cards.
`atlas.search`	A description of a vulnerability type; matching nodes and cards.
`atlas.describe`	A card; the full card and the fixed side of its examples.
`atlas.fetch_examples`	Example identifiers; the vulnerable side of each, within a budget per attempt.
Repository	
`code.list`, `code.grep`, `code.excerpt`	Directory listings, pattern search over source files, and line ranges of a file.
`repo.entrypoints`	The fuzzing entry points with their seed corpora.
`repo.build`, `repo.test`	A build configuration; the build result with new compiler warnings, or the test result with the failing tests.
Probe and execution	
`probe.census`	An entry point and optionally a file; the branch census of Appendix B.2.
`run_input`	An entry point and an input; the exit status and any sanitizer or oracle report on the current build.
`witness.search`	An entry point, seed inputs, and a time limit; fuzzes the injected build and returns inputs that produce a violation.
Editing and verification	
`inject.apply`, `inject.revert`	Exact-match replacements in source files; restore the clean tree.
`oracle.bind`	An invariant, written by the injector or taken from a card, and an expression for each of its roles; the compiled contract oracle, checked to compile on both snapshots.
`verify.run`	An entry point and a witness; evaluates Equation 1 on fresh clean and injected snapshots.
`submit_injection`	The evidence of a passing `verify.run`, a description, and a difficulty rating.
Memory and delegation	
`experience.query`	A repository, entry point, or function; prior sites, failed mutations, and reusable inputs.
`profile.failures`	A CWE; failed trajectories of the student on the feedback set in the latest round (co-evolution only).
`subagent.run`	A question; a report from a subagent with read-only tools.

A subagent receives the atlas, repository, and probe tools but cannot edit, run inputs, or submit. At most 4 subagents run at once, each with a budget of 40 tool calls, and each returns a written report. An attempt ends when the injector submits or gives up, or when it exhausts a budget of 3 hours of wall-clock time. Builds are limited to 30 minutes, the regression suite to 20 minutes, and a call to `witness.search` to 180 seconds.

B.2 PROBE

The census builds the repository with coverage instrumentation, replays the seed corpus of an entry point without fuzzing, and exports branch coverage for each condition and each subcondition of a compound condition. A condition is a guard if one of its outcomes leaves the guarded operation unexecuted, through an early return, a jump to an error path, or a skipped block. The census classifies each guard as load-bearing if some seed reaches it and some seed takes the rejecting outcome, dead if seeds reach it but none is rejected, and unreached otherwise. For each guard it also returns the seeds that take each outcome, which seed the witness search, and features that bear on difficulty: the number of guards that still stand between the entry point and the guard, the depth of the guard in the call chain, and the number of seeds that reach it. Figure 4 shows an excerpt.

Excerpt of a census for the JSON number parser of Appendix C.5

```
cJSON.c:323  guard: (i < sizeof(number_c_string) - 1) && can_access_at_index(input_buffer,
     i)
   reached by 36 number tokens over 22 seeds (18 of the 22 seeds contain a number)
   subcondition 1  i < capacity (64 - 1):    loop-exit cause 0 times    dead
   subcondition 2  input still holds byte i:  loop-exit cause 12 times   load-bearing
   24 tokens instead end at a non-number character; longest number token 20 < 63
```

Figure 4: Excerpt of a probe report for the parser of Appendix C.5. The overflow guard, subcondition 1, is reached by every number but rejected none of the seeds, so the witness search must construct a long number.

B.3 CONTRACT ORACLE

An invariant is written in a small expression language over its roles: integer literals, arithmetic, comparisons, conjunction, disjunction, and negation, without function calls. The injector binds each role to a C expression that is in scope at the operation that requires the invariant, and `oracle.bind` compiles the predicate into a check placed immediately before that operation. Roles are evaluated as 128-bit integers, so the check itself cannot overflow. A violated check prints the contract and the values of its roles and calls `abort`, so the fuzzing engine records the input as a crash together with its stack, and the witness search needs no separate budget for oracle violations. The same instrumentation is applied to both snapshots, and the clean snapshot must replay all seeds and the witness without a violation, which rejects a check that is misplaced or that the original program already violates. The instrumented builds exist only inside the verifier; the public task contains the source without the check. Figure 5 shows a compiled check.

B.4 VERIFICATION AND DEDUPLICATION

`verify.run` builds both snapshots from scratch in the same configuration and replays the witness 3 times on each. The violation must occur on every replay of the injected snapshot and on none of the clean one; a timeout, a failed build, or an infrastructure error makes the verification fail. For a sanitizer witness, this is the check that ARVO, CyberGym, and SEC-bench apply to disclosed vulnerabilities (Mei et al., 2026; Wang et al., 2026; Lee et al., 2025), with the clean snapshot in place of the fixed version. The regression suite runs on both snapshots and excludes the tests that were unreliable at admission. After an injection is accepted, the harness fuzzes the injected build from the witness for 10 minutes and keeps up to 16 further inputs that also satisfy the differential criterion, preferring inputs with distinct stacks. These witness variants are hidden and are used to score repairs. Two accepted environments are duplicates if they are in the same repository and their normalized diffs are identical, or if they apply the same variant of the same card to the same guard; only the first is kept.

After each attempt, the harness adds the attempt's outcome to the experience bank of the repository: the site and card of an accepted injection, the mutations that failed and why, such as no witness within the budget, a failing regression test, or a violation on the clean snapshot, and inputs that reached the site. Later attempts on the same repository query this record, so they avoid sites that are already used or known to fail.

C Dataset and CyberCraft-Bench

C.1 Repository Admission

Each repository is turned into a project in the OSS-Fuzz format: a container definition, a build script that honors the standard compiler and sanitizer flags, and a project description. Where a source provides no fuzzing entry point, an agent writes one against an API that consumes external input, such as a parser, a codec, or a protocol handler, calling it only as the documentation, headers, and existing tests use it; a command-line tool that reads a file can also serve as an entry point. The agent may edit build files but not the project's source code, and it has at most three attempts per repository. Admission is decided by the checks in Table 5, run by the harness. Repositories from Mayhem enter only the training and feedback sets. Table 6 gives the number of repositories that pass each stage. Injection stopped once each group reached its number of environments, so not every admitted repository was injected.

Table 5: Admission checks for a repository. Every check is executed by the harness.

Check	Requirement
Version	Pinned upstream commit, or pinned source package for distribution sources.
Exclusion	Not a project of an evaluation benchmark, nor a fork, embedded copy, or distribution package of one; one repository per upstream project.
Build	Builds with AddressSanitizer, UndefinedBehaviorSanitizer, and coverage instrumentation, each within 30 minutes.
Entry point	At least one entry point whose seeds, replayed on the coverage build, reach project code outside the entry point.
Clean baseline	All seeds replay on the clean build without a sanitizer report, and a short fuzzing run of 10 minutes per entry point finds none.
Tests	If the repository has tests, they run once on the clean build; tests that do not pass reliably are excluded from the regression suite.

Table 6: Repository funnel by source.

Source	Candidates	Admitted	Injected	$\geq$1 environment
OSS-Fuzz	900	620	470	250
ClusterFuzzLite	90	60	45	24
Mayhem	205	150	120	55
Debian	1,400	830	500	220
vcpkg and Conan	1,200	640	437	164
Total	3,795	2,300	1,572	713

C.2 Metadata

Table 7 lists the metadata of an environment and where each field comes from. Only the description and the difficulty rating are reported by the injector; the other fields are derived by the harness from the run.

Distance. The harness symbolizes every stack in the report of the witness: the crash stack and, where present, the allocation, free, and origin stacks, or the stack at the contract check. It removes the frames of the sanitizer runtime, the fuzzing engine, the entry point, and system libraries, which leaves frames in project code. On the crash stack, frame 0 is the innermost remaining frame, and d is the index of the first frame that lies in an edited function. If no edited function is on the crash stack but one is on another stack of the report, the environment is upstream; if none appears on any stack, it is off-stack. An edit outside any function body, such as a changed macro or constant, is attributed to the functions that use it.

Difficulty. The injector rates each environment easy, medium, or hard from what it observed while building it: how deep the injected site sits below the entry point, how many other guards stand on the

Table 7: Metadata recorded for each environment.

Field	Derivation
Mechanism, variant	The card that the injector opened, matched against the shape of its diff; environments whose diff matches no variant of an opened card are marked unmatched.
CWE	The atlas node of that card.
Crash class	The category of the sanitizer report, such as a heap buffer overflow on read; none if only the contract oracle reports.
Witness type	Sanitizer or contract oracle.
Edited functions	The functions whose bodies the diff changes.
Distance	Computed from the report and the edited functions, as described below.
Witness inputs	The accepted witness and its variants, used to score repairs.
Description	The injector's description of the vulnerability, given to the teacher, to the student on the feedback set, and in reinforcement learning (Section 3.5).
Difficulty	The injector's rating as easy, medium, or hard.

path to it, the size of the repository, and whether the operands of the weakened guard are computed in a different function from the one that uses them. The rating is made without solving the environment. Section 4.2 reports repair success by this rating and finds that it orders the environments by measured difficulty; the feedback set holds 100 environments of each rating so that the student profile is read at every level.

C.3 Public Task and Scoring

The public task of an environment contains the repository at the injected revision without its version history, the build script, the fuzzing entry points with their seed corpora, the regression tests, and the objective. The hidden bundle contains the clean revision, the diff, the witness and its variants, the instrumentation of the contract oracle, and the metadata. A solver leaves its answer in a submission directory: the file and function that contain the vulnerability, an input file with the entry point that it is passed to, and a patch. The hidden verifier checks the endpoints in order and stops at the first that fails.

Localization. The named function is one of the edited functions; for a function name that occurs more than once, the file must match as well.

Reproduction. Replayed 3 times through the named entry point, the input produces the designated violation on the injected build and never on the clean build. The violation is designated by the witness type: the same contract for an oracle, and the same crash class for a sanitizer.

Repair. The patch changes only source files of the project, not tests, build scripts, or entry points. After it is applied to the injected revision, the repository builds, the regression tests pass, and neither the witness, its variants, nor the solver's own input produces the violation. The seeds and tests that reached the injected site before the patch must still reach it, which is checked on a coverage build.

The witness passes the reproduction check and reverting the injection passes the repair check, so every environment is solvable at all three endpoints. Because the witness variants are hidden, a patch that blocks only the inputs the solver has seen fails when a variant still produces the violation. With these variants and the reachability requirement, the repair check is stricter than those of the benchmarks of real vulnerabilities in Table 11, which test a patch against a single proof of concept and, in CyberGym-E2E, the project's tests.

A verification that fails for an infrastructure reason, such as a container that does not start, is repeated until it completes, so every attempt is scored on the solver's own outcome and success rates are computed over all environments. During reinforcement learning, such a rollout is instead masked from the loss.

C.4 Statistics

Table 8 describes the splits.

Table 8: Composition of the dataset. Distance and witness type are given as percentages of the environments in each split. The two training runs draw from the same repositories, so the repository counts do not sum to the total, which counts distinct repositories.

Split	Environments	Repositories	CWEs	Local	Upstream	Off-stack	Oracle only
Co-evolution training	3,300	560	381	29.8	35.2	35.0	14.5
Static training	4,000	560	407	31.1	34.6	34.3	13.1
Feedback	300	75	143	30.0	35.3	34.7	13.7
CyberCraft-Bench	379	78	98	30.1	35.1	34.8	14.2
Total	7,979	713	514	30.4	34.9	34.7	13.8

C.5 Example Environment

This environment applies variant V1 of the card in Figure 2 to cJSON, a C library that parses JSON, at release v1.7.18. Table 9 lists its metadata and Figure 5 its injection, contract oracle, and witness. The function `parse_number` copies the characters of a JSON number one at a time into a 64-byte stack buffer and then hands the buffer to `strtod`. One loop guards the copy, and its guard is compound: the loop stops when the index reaches the capacity of the buffer, and also when the input runs out.

Injection. The probe reported this guard as load-bearing, but only through its second term: across 22 seed documents the loop was entered for 36 number tokens and ended 12 of them because the input ran out, while its capacity term was dead and rejected none of them, because the longest number in the corpus was 20 characters and the buffer holds 63 (Appendix B.2). The injector bound the roles of the card to this code, with `dst` as the buffer, `dst_cap` as its size, `src` as the input text, `src_avail` as the bytes that remain, `len` as the loop index, and `written` as the index plus one at each write into the buffer, including the terminating byte, and it applied variant V1 by deleting the capacity term from the guard. The loop then copies until the input runs out, so a JSON number longer than the buffer overflows it. A number of 63 characters or fewer behaves exactly as before, so the change is invisible on ordinary inputs.

Verification. The witness is a JSON document of 200 digits. On the clean snapshot the loop stops at the capacity term and the parser returns normally, and AddressSanitizer and UndefinedBehaviorSanitizer stay silent. On the injected snapshot the copy writes past the buffer, and both sanitizers report the overflow in `parse_number`. The verifier also compiled the invariant `written <= dst_cap` into a contract oracle placed before each write into the buffer and evaluated it on both snapshots with no sanitizer present: it is silent on the clean snapshot and on every seed, and on the injected snapshot it reports `written(65) <= dst_cap(64)` and aborts. Both snapshots build and parse all 22 seeds, so Equation 1 holds. Here the destination is a stack buffer, so a sanitizer already detects the overflow and the oracle agrees with it. The cases in which only the oracle reports are those in which the destination is a field followed by other fields of the same object, as in the examples of Figure 3, and Appendix C.6 walks through such an environment.

Scoring. The overflow occurs in `parse_number`, the edited function, so the environment is local. For localization a solver must name `parse_number`. Any input with a run of 64 or more number characters reproduces the violation, and 63 is the longest run that does not. A repair passes if the library builds, the 22 seeds still parse, and the witness no longer overflows the buffer. Restoring the capacity term passes, and so does any bound that keeps the written length within the buffer. A patch that stops parsing numbers removes the violation but fails the seeds.

Injection in `parse_number`, `cJSON.c` (hidden)

```
     unsigned char number_c_string[64];
     ...
     /* copy the number into a temporary buffer ... */
-    for (i = 0; (i < (sizeof(number_c_string) - 1))
-                && can_access_at_index(input_buffer, i); i++)
+    for (i = 0; can_access_at_index(input_buffer, i); i++)
     {
         switch (buffer_at_offset(input_buffer)[i])
         {
             case '0': ... case '9': case '+': case '-': case 'e': case 'E':
                 number_c_string[i] = buffer_at_offset(input_buffer)[i];
                 break;
             ...
             default:
                 goto loop_end;
         }
     }
 loop_end:
     number_c_string[i] = '\0';
```

Contract oracle compiled from the invariant, inserted by the verifier (never in the public task)

```
+                CC_CONTRACT(i + 1 <= sizeof(number_c_string));  /* written <= dst_cap */
                 number_c_string[i] = buffer_at_offset(input_buffer)[i];
                 break;
     ...
 loop_end:
+    CC_CONTRACT(i + 1 <= sizeof(number_c_string));
     number_c_string[i] = '\0';
```

Witness input and observations

```
witness.json   a JSON number of 200 digits (minimal trigger: 64; 63 is safe)

clean build      parses normally; AddressSanitizer, UBSan, and the oracle silent
injected build   AddressSanitizer: stack-buffer-overflow, WRITE of size 1
                 in parse_number (cJSON.c); UBSan: index 64 out of bounds
                 contract oracle: written(65) <= dst_cap(64), abort
```

Figure 5: Injection, contract oracle, and witness of the example environment. The full environment, with both snapshots, the seeds, and a script that rebuilds them and checks the criterion, is released with the paper.

Table 9: Metadata of the example environment, kept in its hidden bundle. The public task contains the injected repository and the entry point; the description view of reinforcement learning (Section 3.5) adds the description.

Field	Value
Repository	DaveGamble/cJSON, C, release v1.7.18
Entry point	`LLVMFuzzerTestOneInput` in `fuzz_json.c`
CWE	CWE-806, Buffer Access Using Size of Source Buffer
Mechanism	Card of Figure 2, variant V1
Edited function	`parse_number` in `cJSON.c`
Witness type	AddressSanitizer and UndefinedBehaviorSanitizer; the contract oracle agrees
Observed crash class	Stack-buffer-overflow (write)
Distance	Local
Difficulty rating	Medium
Description	The loop that copies a JSON number into a fixed 64-byte buffer is bounded only by the input that remains, not by the size of the buffer. A number of 64 or more characters is written past the end of the buffer.

C.6 A Violation Only the Contract Oracle Detects

The environment of Appendix C.5 is witnessed by a sanitizer. This one is not, and it shows why the contract oracle is needed. It applies variant V1 of the same card to the CoAP message parser of Wakaama, a widely used C implementation of the Lightweight M2M protocol for constrained devices. Figure 6 shows the injection, the oracle, and the witness. As it parses a datagram, `coap_parse_message` copies the message token into `token`, an 8-byte array that is a field of the packet object and is followed in that object by the fields that the rest of the parse fills in. The token length is taken from the Token Length (TKL) field of the header, which is 4 bits wide and so cannot exceed 15. The correct code rejects the reserved lengths 9 to 15 before the copy.

Injection and verification. The injector applied variant V1 by deleting the check that rejects those lengths, keeping the check that the token bytes are actually present in the datagram, so that only the destination side of the copy overflows. The witness is a datagram with TKL set to 15 and a 15-byte token. On the clean snapshot the check rejects the reserved length and the parser returns an error. On the injected snapshot the copy writes 7 bytes past `token`, into the fields that follow it in the same packet object. The write stays inside the object, so AddressSanitizer, which the reconstruction pipelines of Section 2 use to confirm a vulnerability, reports nothing, and a criterion that accepts only sanitizer crashes would discard this injection. UndefinedBehaviorSanitizer is also silent, because the copy is a `memcpy` and not an array subscript whose bound it could check. The contract oracle compiled from `written <= dst_cap` reports `written(15) <= dst_cap(8)` on the injected snapshot and stays silent on the clean snapshot and on every seed, so Equation 1 holds through the oracle alone. This bound is the one the correct code enforces, the check the injection removed, not a property written for the oracle. Because TKL is only 4 bits, 15 is the largest token any datagram can carry, so the overflow is at most 7 bytes and no input can push the write past the end of the object: the sanitizers' silence is a property of the weakness, not of this witness. The overflow is in the edited function, so the environment is local; its interest is the witness type, not the distance.

Injection in `coap_parse_message`, `er-coap-13.c` (hidden)

```
 typedef struct { ...
     uint8_t token[8];       /* dst; COAP_TOKEN_LEN */
     uint8_t accept_num;     /* follows dst in the same packet object */
 ... } coap_packet_t;

     coap_pkt->token_len = TKL(coap_pkt->buffer[0]);        /* 4-bit field, so 0..15 */
-    if (coap_pkt->token_len > COAP_TOKEN_LEN) goto error;  /* reject reserved 9..15 */
     if (current_option + coap_pkt->token_len > data_end) goto error; /* bytes present? */
     memcpy(coap_pkt->token, current_option, coap_pkt->token_len);
```

Contract oracle compiled from the invariant, inserted by the verifier (never in the public task)

```
     if (current_option + coap_pkt->token_len > data_end) goto error;
+    CC_CONTRACT(coap_pkt->token_len <= COAP_TOKEN_LEN);  /* written <= dst_cap */
     memcpy(coap_pkt->token, current_option, coap_pkt->token_len);
```

Witness input and observations

```
witness.bin   one datagram, TKL=15 (the largest a 4-bit field can encode), 15-byte token

clean build      TKL 9..15 rejected as reserved; no overflow; sanitizers and oracle silent
injected build   7 bytes written past token[8] into the following fields, same object
                 AddressSanitizer: silent (the overflow stays inside the object)
                 UndefinedBehaviorSanitizer: silent (memcpy, not a bounded subscript)
                 contract oracle: written(15) <= dst_cap(8), abort
no larger input  TKL is 4 bits, so no datagram can push the write out of the object
```

Figure 6: Injection, contract oracle, and witness of the oracle-only example environment. The full environment, with both snapshots, the seeds, and a script that rebuilds them and checks the criterion, is released with the paper.

C.7 An Off-Stack Environment

The two environments above are local: the overflow is in the edited function. This one is off-stack, and it shows what the distance of Section 3.4 measures. It uses LodePNG, a widely used single-file C library for PNG, at version 20260119. Figure 7 shows the injection and the crash. A PNG palette holds at most 256 colours, so the format allows an indexed image only at bit depths 1 to 8 and an index is at most 255. LodePNG keeps the palette in a fixed 1024-byte buffer of 256 four-byte entries and, in the colour-conversion stage, reads `palette[index]` with no bounds check, relying on the bit depth to keep the index below 256.

Injection and verification. The check that enforces this, `checkColorValidity`, runs while the header is read and rejects an indexed image at bit depth 16. The injector weakened it to accept bit depth 16 as well. The witness is a small indexed image declared at bit depth 16 whose pixels use the index 256. On the clean snapshot the header check rejects it and decoding stops with an error. On the injected snapshot the header is accepted, the check returns, decoding proceeds, and the conversion stage reads one entry past the palette. AddressSanitizer reports a heap-buffer-overflow in `getPixelColorsRGBA8`, and the edited function `checkColorValidity` is on no stack of the report: it validated the header and returned long before the read, so the environment is off-stack. For localization a solver must name `checkColorValidity`, where the malformed input was admitted, not the conversion function where the program crashes. The index 255 is the last that does not overflow, and a larger index gives a wild read. Both snapshots build and decode the seed images, so Equation 1 holds. Like the environment of Appendix C.5, this one is witnessed by a sanitizer; its distance is off-stack rather than local.

Injection in `checkColorValidity`, `lodepng.c` (hidden)

```
 static unsigned checkColorValidity(LodePNGColorType colortype, unsigned bd) {
   switch(colortype) {
     ...
-    case LCT_PALETTE: if(!(bd==1 || bd==2 || bd==4 || bd==8         )) return 37; break;
+    case LCT_PALETTE: if(!(bd==1 || bd==2 || bd==4 || bd==8 || bd==16)) return 37; break;
     ...
   }
   return 0; /* allowed colortype/bitdepth combination */
 }
```

Witness input and observations

```
witness.png   indexed image, bit depth 16, pixels using index 256 (one past the palette)

clean build      checkColorValidity returns error 37 while reading the header; no crash
injected build   AddressSanitizer: heap-buffer-overflow, READ, 0 bytes right of a
                 1024-byte region allocated in readChunk_PLTE
                   #1 getPixelColorsRGBA8   (colour conversion)
                   #2 lodepng_convert
                   #3 lodepng_decode
                 checkColorValidity appears on no stack of the report  ->  off-stack
```

Figure 7: Injection and crash of the off-stack example environment. The guard that admits the malformed header is in a function that has returned before the overflow, so the edited function is on no stack of the report. The full environment and a script that rebuilds it and checks the criterion are released with the paper.

C.8 An Upstream Environment

The environment of Appendix C.5 is local and the one of Appendix C.7 is off-stack. This one is upstream: the edited function is on the crash stack, but below the frame that crashes. It applies the mechanism of the card in Figure 2 to miniz across a function boundary. The public function `mz_zip_reader_extract_to_mem` extracts an archive entry into a caller-supplied buffer. Its worker, `mz_zip_reader_extract_to_mem_no_alloc1`, reads the entry's uncompressed size from the archive, checks that the caller's buffer is at least that large, and then calls a read helper that copies the bytes in.

Injection and verification. The injector deleted the capacity check, so the length the copy uses is the one the archive declares, not the size of the destination. The harness extracts each entry into a fixed 64-byte buffer, and the witness is an archive with one stored entry of 200 bytes. On the clean snapshot the check rejects the entry with a buffer-too-small error. On the injected snapshot the read helper `mz_zip_mem_read_func` copies 200 bytes into the 64-byte buffer, and AddressSanitizer reports a heap-buffer-overflow in that helper. The edited function is one frame below the helper on the crash stack, so the distance is upstream: a solver that reads only the top of the stack sees the copy helper, which is correct on its own, and has to follow the call back to the caller whose check was removed. For localization it must name that caller, not the helper where the write happens. An entry of 64 bytes fits and 65 is the first that overflows. Both snapshots build and extract the seed entries, so Equation 1 holds.

Injection in `mz_zip_reader_extract_to_mem_no_alloc1`, `miniz_zip.c` (hidden)

```
     /* Ensure supplied output buffer is large enough. */
     needed_size = (flags & MZ_ZIP_FLAG_COMPRESSED_DATA) ? file_stat.m_comp_size
                                                         : file_stat.m_uncomp_size;
-    if (buf_size < needed_size)
-        return mz_zip_set_error(pZip, MZ_ZIP_BUF_TOO_SMALL);
     ...
     /* stored entry: copy needed_size bytes into pBuf through the read callback */
     pZip->m_pRead(pZip->m_pIO_opaque, cur_file_ofs, pBuf, (size_t)needed_size);
```

Witness input and observations

```
witness.zip   one stored entry of 200 bytes; the harness extracts into a 64-byte buffer

clean build      buffer-too-small error; no crash
injected build   AddressSanitizer: heap-buffer-overflow, WRITE of size 200,
                 0 bytes to the right of a 64-byte region
                   #2 mz_zip_mem_read_func                     (copy helper: crash frame)
                   #3 mz_zip_reader_extract_to_mem_no_alloc1   (edited function: below it)
                 the edited function is on the stack, below the crash  ->  upstream (d>0)
```

Figure 8: Injection and crash of the upstream example environment. The removed check is in the caller; the overflow is in the read helper it calls, so the edited function sits below the crash frame on the stack. The full environment and a script that rebuilds it and checks the criterion are released with the paper.

D Training Details

D.1 Teacher Demonstrations

The teacher runs in the Codex agent scaffold with DeepSeek-V4-Flash, in a container that holds the public task and can reach only the model API. It receives the blind prompt of Appendix F with the description of the vulnerability added, and each attempt is limited to 90 minutes of wall-clock time, 40 agent turns, and 300 tool calls. The hidden verifier scores an attempt once, after the agent exits, and only attempts that pass all three endpoints are kept. Each environment receives at most six attempts. An environment with fewer than three accepted trajectories after six attempts is replaced by a new environment from the same run, so every training environment has three trajectories.

Each accepted episode is converted into a transcript of the student's chat format: the system prompt and the task as the first turns, then the assistant's messages and tool calls, each followed by the output of the tool. Tool outputs longer than 8,192 tokens are truncated in the middle, as the scaffold truncates them for the agent. The teacher's reasoning is not included, because the scaffold exposes only summaries of it. The loss is computed on the assistant turns only.

D.2 Supervised Fine-Tuning

Table 10 lists the hyperparameters of both stages. Every supervised checkpoint, including each round of co-evolution, starts from the base model and uses the same settings.

D.3 Reinforcement Learning

Reinforcement learning is applied to the co-evolution run only (Section 4.1); the static run is compared after supervised fine-tuning.

Views. A view is defined by what the task text adds to the blind prompt (Appendix F) and by the milestones it leaves open. The blind and description views leave all four open. The reproduction view adds the sanitizer or oracle report of the witness, without the input, and leaves reach and reproduction open. The repair view adds the witness input and its report and leaves only repair open; the witness variants remain hidden. An environment starts in the blind view if the teacher solved it in at least 2 of its first three attempts, and in the description view otherwise. It moves to

the next view when the screened half of a group fails entirely, and leaves the pool when the screened half succeeds entirely or when the repair view fails. Over the 100 steps, 24%, 44%, 20%, and 12% of the completed groups of the co-evolution run were in the blind, description, reproduction, and repair views, and 41% of screened halves were uniform and not completed.

Reward. The weights of Equation 2 are $w_{\mathrm{loc}} = 0.15$, $w_{\mathrm{reach}} = 0.1$, $w_{\mathrm{poc}} = 0.25$, and $w_{\mathrm{rep}} = 0.5$. They are not renormalized within a view, so a rollout in the repair view earns at most w_{rep}, and the advantage is computed within the group, whose rollouts share one view. The gate checks that the patch touches no test, build script, entry point, or verifier file, and that the seeds and tests that reached the injected site before the patch still reach it.

Optimization. Advantages are group-relative, with the group mean as the baseline and without division by the standard deviation. The policy loss is clipped with an asymmetric range and averaged over tokens. Rollouts are generated asynchronously with vLLM while the environments run in containers on the CPU cores of the same node, with at most 64 containers at a time, and a rollout may lag the trained policy by at most one step. The token identifiers and log-probabilities are recorded as the inference engine samples them, so the trainer scores exactly the sampled tokens rather than a re-tokenized transcript. The remaining mismatch between the probabilities of the inference engine and the trainer, and the lag, are corrected with a truncated importance weight per token, capped at 2. A rollout ends when the agent exits or reaches 100 turns, 64k tokens of context, or 60 minutes.

Table 10: Hyperparameters.

	Supervised fine-tuning	Reinforcement learning
Initialization	Qwen3.5-9B base	Last supervised checkpoint
Parameters updated	All	All
Optimizer	AdamW, $\beta = (0.9, 0.95)$, weight decay 0.1	AdamW, $\beta = (0.9, 0.95)$, weight decay 0.1
Learning rate	1×10^{-5}, cosine, 3% warm-up	1×10^{-6}, constant
Epochs or steps	3 epochs	100 steps
Batch	64 transcripts	32 environments $\times$ 8 rollouts
Maximum length	65,536 tokens	65,536 tokens, 100 turns
Clipping	None	$\epsilon_{\mathrm{low}} = 0.2$, $\epsilon_{\mathrm{high}} = 0.28$
KL penalty	None	0.001 to the supervised checkpoint
Sampling	None	Temperature 1.0, top-p 1.0
Screening	None	4 of 8 rollouts first
Precision and hardware	bf16, 8 NVIDIA H200	bf16, 8 NVIDIA H200

D.4 Student Profile

After each round, the harness aggregates the student's attempts on the feedback set by CWE, and the injector writes a report from this table and the failed trajectories, using the prompt in Appendix F. The report enters the injector's instructions in the next round, and the trajectories remain available through `profile.failures`. Figure 9 shows an excerpt.

Excerpt of a student profile after round 2

CWE	Attempts	Localize	Reproduce	Repair
CWE-787 Out-of-bounds Write	31	0.81	0.58	0.52
CWE-806 Buffer Access Using Size of Source Buffer	6	0.33	0.00	0.00
CWE-191 Integer Underflow	9	0.56	0.22	0.11

Report. The student localizes and repairs bounds errors whose overflow is reported by AddressSanitizer at the edited function. It fails where the violation is silent or occurs after the edited function has returned. On all six CWE-806 environments it located the copy but not the weakened check, and it fuzzed without a crash and stopped, because the overflow stayed inside an object. Its integer underflows fail at reproduction: it finds the subtraction but does not construct a length field smaller than the header. Mechanisms that separate a check from the use of its result, or that corrupt a field of the same object, are the most informative to generate next.

Figure 9: Excerpt of a student profile.

E Evaluation Protocols

Table 11 summarizes what the agent receives and how success is decided on each benchmark. In every case, success is decided by the benchmark's own checks, run after the agent exits.

Table 11: Evaluation protocols. Tasks are counted in the release we use; the success check is the benchmark's own.

Benchmark	Tasks	Agent input	Success check
CyberCraft-Bench	379, C/C++	Injected repository, build and test scripts, fuzzing entry points and seeds, generic objective	Nested endpoints of Section 3.4, run by the hidden verifier
SEC-bench (CVE split)	200, C/C++	PoC task: the sanitizer report. Patch task: the bug report and a reproducing input	PoC task: the sanitizer report is reproduced. Patch task: the project builds and the input no longer triggers the sanitizer
PatchEval (Verified)	230; Go, JavaScript, Python	Repository and CVE description	The vulnerability's proof-of-concept test passes
CyberGym-E2E (patch-only)	920, C/C++	Source, ground-truth proof of concept, and its crash log	Project tests pass and the proof of concept no longer crashes the program
SWE-bench Verified	500, Python	Repository and issue	All fail-to-pass and pass-to-pass tests pass

Scaffold. All models run in the Codex agent scaffold (Codex CLI 0.31) with its default shell and file-editing tools, inside the container that each benchmark provides. The agent's network access is limited to the model API, and web search is disabled. GPT-5.5 runs with reasoning effort high; Claude Opus 4.8, DeepSeek-V4-Flash, and Kimi K3 are queried through their providers' APIs with default settings, and the Qwen3.5-9B checkpoints are served with vLLM at temperature 0.2. Each task has a limit of 60 minutes of wall-clock time and 100 agent turns; an agent that reaches the limit is scored on what it has produced. A run that fails for an infrastructure reason, such as a container that does not start or an exhausted API quota, is repeated, so that a task is scored as failed only on the agent's own outcome.

Decontamination. The projects of SEC-bench, PatchEval, and CyberGym-E2E are removed from the repository pool before injection, together with their forks, embedded copies, and distribution packages, which we match by upstream repository and package name. The fix commits of their vulnerabilities are removed from the atlas sources, by CVE identifier for SEC-bench and PatchEval and by fix commit for CyberGym-E2E. For CyberGym-E2E, we use a harness version whose repositories contain no history beyond the vulnerable revision, so the fix cannot be recovered with `git`. SWE-bench Verified contains only Python repositories and shares no project with the pool.

F AGENT PROMPTS

The prompts below are abridged; tool descriptions, output formats, and examples are omitted.

Injector (system prompt, abridged)

You are given one real, currently correct C/C++ repository. Make one function in it contain a genuine vulnerability, one that a competent engineer could plausibly have written, and prove that it is real before you submit it.
Read the repository first. Then browse or search the atlas for a mechanism whose recognition conditions match code you have read, and study its examples; they teach the shape of the mechanism, not code to copy. Use the probe to find checks that the seeds reach; a check that they reach but never fail can be weakened without changing their behavior. A guard whose operands are computed in one function and used in another makes a harder and more realistic task; prefer it when it exists, but never fabricate that structure.
Before you edit, write down how a real author could have come to write the weakened code, and make the code consistent with that account: remove or reuse anything that only fed the old check, keep sibling code symmetric, match the style of the file, and keep the diff small. A new compiler warning or a failing regression test rejects the injection. If the card has an invariant, bind its roles and compile the oracle. Search for a witness from the seeds that the probe returned, and run `verify.run`. Submit only with the evidence of a passing verification, together with a description of the vulnerability and your rating of its difficulty.
Consult the experience bank before choosing a site. Giving up is cheap: if no card binds to the code, or no witness appears, revert and try another site or card.
[Co-evolution rounds] The report below summarizes how the student performs on the feedback environments. Use it as a reference when you choose what to inject; `profile.failures` returns the student's failed trajectories. Every injection must still pass verification. *Report:* ...

Atlas mining agent (abridged)

You are given a CVE record and the commit that fixes it, with its diff and surrounding code, and read access to the atlas. Decide what vulnerability mechanism the fix removes: the program roles involved, the property the fixed code maintains, and how the vulnerable code breaks it. Describe the mechanism, not the patch, and in terms that do not depend on this project.
If a card with the same roles and the same mutation exists, add the fix as an example of it. A shared consequence, such as an out-of-bounds read, is not enough. Otherwise create a card under the most specific CWE node that applies, with at least two recognition conditions visible in correct code, an invariant over the roles that can be evaluated on program values, and the variant that this fix shows. For examples, give anchors as text copied exactly from the diff; never write code yourself. If the commit does not fix a vulnerability mechanism, for instance because it only updates versions, tests, or formatting, skip it and say why.

Solver: blind task, used for CyberCraft-Bench

The repository in `/workspace/repo` contains a security vulnerability. It builds with `build.sh`, and `TASK.md` lists its fuzzing entry points, their seed inputs, and how to run its tests. Independently discover, reproduce, and repair the vulnerability. When you finish, leave in `/workspace/submission`: `finding.json`, with the file and the function that contain the vulnerability; an input file and the name of the entry point to which it is passed, which triggers the vulnerability; and `fix.patch`, a patch that removes the vulnerability without breaking the functionality of the project. Do not modify tests, build scripts, or entry points. You may call `validate_submission` to check that your files are well formed and that your input runs, which does not reveal whether the vulnerability is the intended one; call `submit` once when you are done. Your submission is scored after you exit.

Additions for the other views

Description. Description of the vulnerability: ...
Reproduction. Running an input through entry point ... produces the following report. Write an input that produces this report. ...
Repair. The input `witness.bin`, passed to entry point ..., produces the following report. Repair the vulnerability. ...

Student profile report (abridged)

You generate vulnerabilities for training a student agent. Below are the student's results on the feedback environments, by CWE and endpoint, and you can read its failed trajectories. Write a short report of at

most 300 words: where the student fails, at which endpoint, and what in its trajectories explains the failure. Describe the failures in terms of mechanisms and program structure. Do not prescribe quotas; the report is a reference for choosing future injections.

G LIMITATIONS

Resource constraints limit our experiments in two respects. First, all teacher demonstrations come from open-weight models, DeepSeek-V4-Flash and Qwen3.6-27B, not from proprietary models such as GPT-5.5 and Claude Opus 4.8, which score higher on CyberCraft-Bench (Table 1). Second, we train a single student, Qwen3.5-9B. The student already gains more from the stronger of our two teachers (Table 2). Stronger teachers and larger students are left to future work.