SECUREVIBE: MAKING VIBE CODING MORE SECURE

Anonymous authors
Paper under double-blind review

ABSTRACT

As vibe coding becomes increasingly capable and widespread, security vulnerabilities in even functionally correct solutions are a growing concern. When investigating functionally correct but insecure solutions, we find that the insecure agent is less than half as likely to conduct effective planning and testing for the hidden security risks behind the functional requirements. Motivated by this, we develop SECUREVIBE, a training recipe that explicitly targets planning and testing for code security. SECUREVIBE constructs training signals around these security behaviors. It includes supervised fine-tuning on the security suite with 4 security tasks, and post-training methods SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$ to enhance security capabilities from verifiable execution feedback and hint-based self-supervision. Our SECUREVIBE outperforms the baseline on two types of security coding tasks across 4 benchmarks. Specifically, SECUREVIBE improves the security pass@1 by 6.9 points on BaxBench. The gains extend to unseen CWE categories, with improvements of 11.5 points on SusVibes. Meanwhile, it also improves functionality pass@1 by 13.6 points on the security coding task SusVibes and 4.1 points on the generic coding task SWE-bench Verified. Further analysis offers two practical insights: (i) diversifying supervision across security planning, coding, and testing strengthens security behaviors more effectively than adding coding trajectories alone, and (ii) hint-guided supervision is particularly valuable when the agent's existing security capabilities are insufficient to learn effectively from outcome feedback.

1 INTRODUCTION

Large language model (LLM) coding agents can now modify real-world software repositories with increasing success. Given a natural-language request, these agents can inspect a codebase, plan changes, edit multiple files, and execute tests. These capabilities have made "vibe coding" an increasingly practical workflow in which developers describe the desired behavior and rely on an agent to implement it. Progress on benchmarks such as SWE-bench (Jimenez et al., 2024) and Terminal-Bench (Merrill et al., 2026) further shows that coding agents can solve complex software-engineering tasks beyond isolated code completion.

However, functional correctness does not always guarantee security. An implementation may pass its functional tests while still introducing vulnerabilities (Ye et al., 2026; Baumann et al., 2026). These vulnerabilities may result from unsafe input handling, missing authorization checks, insecure API usage, or other security-sensitive decisions. This risk is especially important in vibe coding because users may accept an agent's changes without carefully auditing them. Existing approaches mainly improve functional task completion through better agent scaffolding, supervised fine-tuning, or reinforcement learning (Pan et al., 2025; Ma et al., 2024; Wei et al., 2025a). Secure-code benchmarks usually measure whether generated code contains vulnerabilities (Vero et al., 2025; Zhao et al., 2026b; Chen et al., 2025). However, they rarely explain what capabilities coding agents lack or how agents can acquire them. Therefore, it remains unclear why functionally capable agents produce insecure code and how training can jointly improve secure and general coding capabilities.

We study secure vibe coding through three research questions. *RQ1: What capabilities make a coding agent secure? RQ2: How can an agent acquire these capabilities? RQ3: Do these capabilities generalize to new security settings?*

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

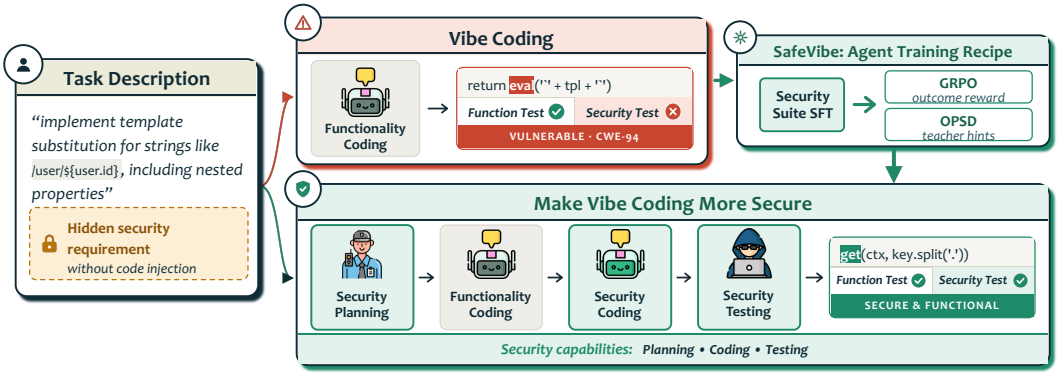

Figure 1: SECUREVIBE: making vibe coding more secure by enhancing security capabilities via a mixture of security tasks and post-training.

For RQ1, we compare the trajectories between the insecure and secure implementations. We find that the insecure agent, although it also provides a functional solution, is less than half as likely to plan and test for the hidden security requirements. Such a lack of security awareness makes them stop after passing functionality tests, ignoring the security risks.

Motivated by these findings, we develop SECUREVIBE, a security training recipe covering data-mixture construction and model training to answer RQ2. As shown in Figure 1, SECUREVIBE$_{base}$ combines *Functionality-Focused Coding, Security-Focused Coding, Security Planning, and Security Testing* to teach agents to recognize security requirements, implement protective controls, and synthesize security tests. We then design **outcome-based SECUREVIBE$_{rl}$** and **hint-guided SECUREVIBE$_{hg}$** to investigate the effectiveness of further post-training. SECUREVIBE$_{rl}$ learns from executable functional and security feedback, while SECUREVIBE$_{hg}$ receives dense token-level supervision from a self-distillation teacher equipped with instance-specific security hints.

To answer RQ3, we train Qwen 3.5 35B with SECUREVIBE on AutoBaxBench (von Arx et al., 2026) and PatchEval-Gen (Wei et al., 2025b) and evaluate its effectiveness on two more benchmarks SusVibes (Zhao et al., 2026b) and BaxBench (Vero et al., 2025). PatchEval-Gen and SusVibes require coding agents to implement features in real-world repositories, while AutoBaxBench and BaxBench ask the agent to implement the repo-level backend from scratch. In the validation set of PatchEval-Gen and AutoBaxBench, SECUREVIBE$_{rl}$ improves average FuncPass by 9.78 percentage points, while SECUREVIBE$_{hg}$ improves average SecPass by 8.23 percentage points over the base model. These gains generalize to SusVibes and BaxBench, where the corresponding improvements reach 10.9 and 4.8 points. On unseen-CWE subsets, SECUREVIBE improves SecPass by 11.5 points on SusVibes and 7.1 points on BaxBench. Security training also strengthens general coding performance: every trained checkpoint outperforms the base model on SWE-bench Verified (Jimenez et al., 2024), with gains of up to 4.1 points.

Further analysis shows that SECUREVIBE elicits planning, coding, and testing for security more frequently than the other recipes, behaviors we identified as essential for secure solutions. It also provides practical guidance for post-training: first, trigger the agent's security behaviors by supervised training with mixed security-related tasks. Then, choose the appropriate post-training methods to further enhance security: use SECUREVIBE$_{rl}$ if the SFT checkpoint always shows great security capabilities for dense positive outcome feedback; otherwise, use SECUREVIBE$_{hg}$ with hint-based teacher guidance to get denser supervision.

Our main contributions are as follows:

- We identify that insecure agents usually lack awareness to plan and test for the implicit security requirements, leading to a performance gap between functionality and security (Section 3.1).
- We introduce SECUREVIBE, a secure-coding training recipe that combines Security Suite SFT with outcome-based SECUREVIBE$_{rl}$ or hint-guided SECUREVIBE$_{hg}$ (Sections 3.2 and 3.3).
- We demonstrate functional and security gains under held-out and cross-benchmark evaluation, generalize to unseen CWE categories, and improve general coding performance (Section 4).

- We analyze the complementary strengths of SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$ and derive practical guidance for selecting post-training methods based on reward density (Section 5).

2 RELATED WORK

Coding Agents and Optimization. LLM-based coding agents can perform repository-level tasks such as bug fixing, feature implementation, test generation, environment setup, and library generation (Jimenez et al., 2024; Mündler et al., 2024; Eliseeva et al., 2025; Zhao et al., 2025). Prior work improves these agents through better actions, workflows, and inference-time scaling (Yang et al., 2024a; Xia et al., 2025; Gao et al., 2025), or through supervised and reinforcement-based model training (Zhang et al., 2025; Ma et al., 2025; Wei et al., 2025a; Kim et al., 2026). These methods primarily optimize functional task completion rather than implementation security. Among general post-training methods, GRPO estimates advantages from relative rewards within groups of responses without training a separate critic (Shao et al., 2024). DAPO improves GRPO through dynamic sampling and asymmetric clipping (Yu et al., 2026), while GSPO moves importance weighting and clipping from the token level to the sequence level (Zheng et al., 2025). By contrast, OPD learns from teacher feedback on trajectories sampled by the student (Song & Zheng, 2026), and OPSD uses the same model as both teacher and student under different contexts (Zhao et al., 2026a). These two families motivate our comparison of reward-based optimization and self-distillation for security-oriented coding agents.

Secure-Code Benchmarks. Various benchmarks have emerged to assess both the security and functional correctness of LLM-generated code. Earlier benchmarks focused on single-turn generation in limited settings, such as a single file or function (Siddiq et al., 2024; Peng et al., 2025; Yang et al., 2024c; Pearce et al., 2025). More recent benchmarks have expanded their scope to repository-level tasks that may require edits across multiple files. BaxBench (Vero et al., 2025) and AutoBaxBuilder (von Arx et al., 2026) evaluate backend application security across popular frameworks and programming languages using functional tests, security tests, and expert-designed exploits. SecureAgentBench (Chen et al., 2025) mine repository-level vulnerability-fixing commits and repurpose them as benchmark tasks. SusVibes (Zhao et al., 2026b) investigates the gap between functional correctness and security across vibe coding agent frameworks. These benchmarks focus primarily on evaluation and provide limited data for training security-aware coding agents.

Secure Code Generation. Prior work improves secure code generation through supervised tuning on secure and insecure examples and reinforcement learning with synthesized security tasks and multi-objective rewards (He et al., 2024; Liu et al., 2026). Related functional code generation methods use test-based critics, reliable execution rewards, and sandboxed evaluation (Le et al., 2022; Liu & Zhang, 2025). Recent work primarily targets model-level code generation. SecCoderX (Wu et al., 2026) applies online reinforcement learning with a learned vulnerability reward model. SRCode uses this data with token-level reinforcement learning rewards (Quan et al., 2026). GoodVibe (Thang et al., 2026) selectively fine-tunes security-relevant neurons to improve secure generation efficiently. Together, these methods advance secure code generation at the model level. In contrast, we develop a security training recipe for executable, repository-level agents that inspect codebases, implement changes, and validate their solutions through tool interaction.

3 SECUREVIBE: A SECURITY TRAINING RECIPE FOR CODING AGENTS

3.1 MOTIVATION: WHAT CAPABILITIES ENABLE SECURE CODING AGENTS?

To investigate which capabilities distinguish secure solutions from functionally correct but insecure ones, we compare GPT 5.6 Sol and Qwen 3.5 35B on SusVibes using the same mini-swe-agent harness (Yang et al., 2024b). Here, GPT 5.6 Sol performs as a more secure agent, while Qwen 3.5 35B is less secure.

By comparing the trajectories of functionally correct solutions among secure and insecure results, we find that the insecure trajectories have less effective planning and testing for the implicit security requirements. Sometimes they have generic security attempts, but cannot address the target vulnerability. To quantify this observation, we annotate security attempts in these trajectories: (i) **Security Planning**, which identifies a security concern and proposes a response; (ii) **Security Coding**, which

implements a security control; and (iii) **Security Testing**, which constructs or executes a check of a security property.

We annotate trajectories that produce functionally correct solutions and examine the correlation between security behaviors and security outcomes. A behavior is target-aligned if it addresses the security property identified by the benchmark's reference evidence. Further annotation details are provided in Appendix A and Appendix B.

Insecure agent shows significantly fewer effective security attempts, leading to poor security performance. We restrict the analysis to tasks for which both models produce functionally correct solutions. As shown in Figure 3, GPT 5.6 Sol exhibits planning, coding, and testing more frequently than Qwen 3.5 35B. We compute the agent differences in behavior and security outcomes for each shared task and correlate these paired differences between tasks. This analysis examines whether a larger behavioral advantage is correlated with a larger security advantage on the same task. The correlations are $r = 0.497$ for planning, $r = 0.652$ for coding, and $r = 0.186$ for testing. Thus, differences in planning and coding show a strong correlation with the security gap.

Relevant security behaviors are positively related to higher security pass rates. We also compare trajectories of the same agent on their secure and insecure solutions. As shown in Figure 2, among functionally correct solutions produced by GPT 5.6 Sol, secure solutions exhibit target-aligned planning, coding, and testing more frequently than insecure solutions. The correlations with the success of the security test are $r = 0.285$ for planning, $r = 0.469$ for coding, and $r = 0.199$ for testing. This indicates that, if we can enhance agents' capabilities in effective planning, coding, and testing for implicit security requirements, we can mitigate the performance gap between high functionality correctness but low security.

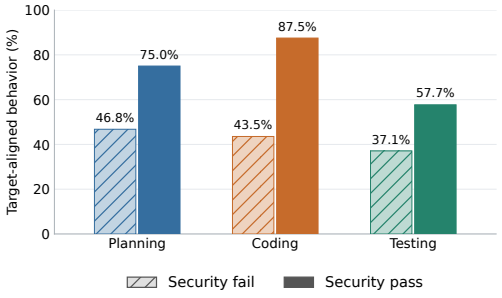

Figure 2: Functionally correct GPT 5.6 Sol solutions. Secure solutions exhibit target-aligned security planning, coding, and testing more frequently than insecure solutions.

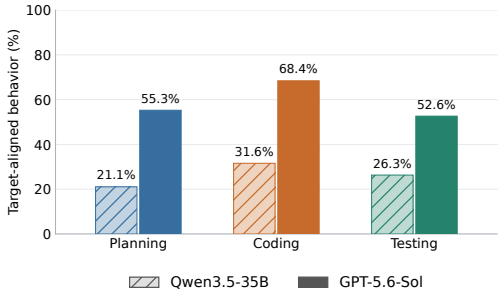

Figure 3: GPT 5.6 Sol and Qwen 3.5 35B on shared functionally correct tasks. GPT 5.6 Sol exhibits target-aligned security planning, coding, and testing more frequently than Qwen 3.5 35B.

3.2 Cold Start: Security Suite for Supervised Fine-Tuning

Motivated by observations, we introduce a **Security Suite** in SecureVibe, which includes four training tasks: *Functionality-Focused Coding, Security-Focused Coding, Security Planning, and Security Testing*. We mix these tasks in our Security Suite for supervised fine-tuning, explicitly enhancing these security capabilities in coding agents.

Functionality-Focused Coding follows the standard task setting: the agent receives a functionality description and implements the requested feature without vulnerability-specific guidance. This represents the common vibe-coding scenario.

Security-Focused Coding augments the agent's prompt with instance-specific guidance, such as the relevant CWE category or CVE description. This guidance makes implicit security requirements explicit and directs the agent's attention toward secure implementation. We derive it from the ground-truth vulnerability metadata provided by each dataset.

Security Planning requires the agent to identify implicit security requirements from the functionality description before implementation. In real-world scenarios, oracle vulnerability information is unavailable, and users may neither recognize nor specify the relevant security concerns. A secure

coding agent must therefore infer these requirements proactively. This task bridges the gap between Security-Focused Coding and Functionality-Focused Coding.

Security Testing requires the agent to synthesize security test cases. Because private security tests are unavailable to the agent, it must learn to validate its own solution before submission. In this training task, the agent receives the feature description and oracle vulnerability information and generates tests that distinguish a vulnerable implementation from a secure one.

3.3 POST-TRAINING: OUTCOME-BASED SECUREVIBE$_{rl}$ AND HINT-GUIDED SECUREVIBE$_{hg}$

To refine behavior on trajectories sampled by the agent itself, we investigate two post-training methods: reinforcement learning SECUREVIBE$_{rl}$ and hint-guided on-policy self-distillation SECUREVIBE$_{hg}$.

Outcome-Based SECUREVIBE$_{rl}$. We initialize SECUREVIBE$_{rl}$ from SECUREVIBE$_{base}$ and use GRPO for training. Let π_θ denote the agent policy. For each prompt x, we sample a group of G agent trajectories $\{o_i\}_{i=1}^{G}$ and assign each a scalar reward r_i using executable feedback on patch validity, functional correctness, and security correctness. A trajectory receives a reward of -1 if it does not produce a patch, -0.75 if the patch cannot be applied, and -0.5 if the applied patch fails both test suites. Functional-only success receives 0.5, while passing both functional and security tests receives 1. Each trajectory's advantage is its reward normalized within the group:

$$\hat{A}_i = \frac{r_i - \text{mean}\left(\{r_j\}_{j=1}^{G}\right)}{\text{std}\left(\{r_j\}_{j=1}^{G}\right)}. \tag{1}$$

which weights the clipped policy-gradient update for every model-generated token of o_i. Following dynamic sampling (Yu et al., 2026), we retain only groups with $\text{std}(\{r_i\}) > 0$ and $\max_i r_i > 0$, i.e., nonzero reward variance and at least one positive outcome, maximizing informative relative-advantage signals. The fine-grained reward further distinguishes intermediate outcomes that a binary secure-success reward would treat identically.

Hint-Guided SECUREVIBE$_{hg}$. We initialize SECUREVIBE$_{hg}$ from the same SECUREVIBE$_{base}$ and apply OPSD on it. The student and teacher models use the same model policy π_θ. To provide security supervision, we create different levels of security hints for SECUREVIBE$_{base}$, such as CWE, CVE description, and attack vector, to enhance its security capabilities (Appendix E). The student receives the standard Functionality-Focused Coding context x, whereas the teacher context $\tilde{x}$ additionally includes a security-specific system prompt and the instance-level hints. On trajectories $\hat{y} \sim \pi_\theta(\cdot \mid x)$ sampled by the student itself, the model distills the teacher's output distribution into the student:

$$\mathcal{L}_{\text{SECUREVIBE}_{hg}}(\theta) = \mathbb{E}_{\hat{y}\sim\pi_\theta(\cdot|x)}\left[\sum_{t=1}^{|\hat{y}|} D\Big(\text{sg}\left[\pi_\theta(\cdot \mid \tilde{x}, \hat{y}_{<t})\right] \,\Big\|\, \pi_\theta(\cdot \mid x, \hat{y}_{<t})\Big)\right], \tag{2}$$

where $\hat{y}_{<t}$ denotes the tokens generated before position t, D is the token-level divergence (e.g., reverse KL), and $\text{sg}[\cdot]$ denotes stop-gradient, so gradients flow only through the student. OPSD extracts dense supervision from every on-policy trajectory, making it particularly effective when executable rewards are sparse.

4 EXPERIMENTS

Benchmarks. We choose two kinds of repository-level coding tasks with security requirements. PatchEval-Gen and SusVibes (Zhao et al., 2026b) evaluate repository-level feature implementation in existing codebases, whereas AutoBaxBench (von Arx et al., 2026) and BaxBench (Vero et al., 2025) require agents to build web applications from scratch. We use AutoBaxBench and PatchEval-Gen as our training set. AutoBaxBench contains 560 tasks with 6 programming languages and 9 CWE categories. PatchEval-Gen contains 200 tasks with 3 languages and 39 CWE categories. We curate PatchEval-Gen from PatchEval (Wei et al., 2025b) using the pipeline proposed by SusVibes (Zhao et al., 2026b). To prevent evaluation overlap, we remove all PatchEval-Gen instances that also appear in SusVibes. BaxBench contains 392 tasks across 6 languages and 13 CWE categories. SusVibes contains 186 Python tasks covering 76 CWE categories. We additionally evaluate on the 500-task SWE-bench Verified split (Jimenez et al., 2024) to assess general software-engineering performance. Benchmark details are provided in Appendix C. We report FuncPass, Pass@1 on the functional tests,

and SecPass Pass@1 for solutions that pass both functional and security tests. All results are based on the mean and standard deviation over 3 runs.

Training Configurations. All experiments use Qwen 3.5 35B (Qwen Team, 2026) as the base model and mini-swe-agent (Yang et al., 2024b) as the harness. We compare three SFT data recipes. *Security Only* uses distilled secure trajectories from the training instances. *Simple Mixture* contains 1,670 examples, including 544 functional and 1,126 security examples. SECUREVIBE$_{sft}$ with *Security Suite* contains 1,648 examples (505 functional, 938 security, 110 security-test synthesis, and 95 security-planning trajectories). The similar sizes of Simple Mixture and Security Suite allow us to isolate the effect of explicitly teaching security planning and testing.

We train each SFT configuration for 3 epochs. The training data for all configurations include instances from both PatchEval-Gen and AutoBaxBench. For SFT, we use a global batch size of 32 and a learning rate of 10^{-5}. For SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$, each batch contains 4 prompts with 8 samples per prompt, leading to a global batch size of 32 samples per optimization step. We limit the maximum sequence length to 128K tokens. All experiments are conducted on 8 NVIDIA B200 GPUs. More details are in Appendix E.

4.1 EXPERIMENTAL RESULTS

Table 1: Results for SFT data recipes and post-training methods on the held-out validation set of PatchEval-Gen and AutoBaxBench. Δ denotes the difference from the Qwen 3.5 35B baseline. Results are averaged over 3 repetition runs.

Benchmark	Model	FuncPass	ΔFuncPass	SecPass	ΔSecPass
PatchEval-Gen (100)	Qwen 3.5 35B	$58.33_{\pm 4.04}$	–	$13.33_{\pm 1.53}$	–
	Security Only SFT	$62.00_{\pm 2.00}$	+3.67	$9.33_{\pm 1.53}$	−4.00
	SECUREVIBE$_{rl}$	$64.00_{\pm 3.61}$	+5.67	$12.67_{\pm 2.52}$	−0.66
	Simple Mixture SFT	$54.67_{\pm 2.31}$	−3.66	$10.00_{\pm 3.00}$	−3.33
	SECUREVIBE$_{base}$	$62.00_{\pm 1.00}$	+3.67	$12.67_{\pm 1.15}$	−0.66
	SECUREVIBE$_{rl}$	$60.67_{\pm 3.51}$	+2.34	$14.00_{\pm 1.00}$	+0.67
	SECUREVIBE$_{hg}$	$60.00_{\pm 2.65}$	+1.67	$15.33_{\pm 0.58}$	+2.00
AutoBaxBench (420)	Qwen 3.5 35B	$33.73_{\pm 0.14}$	–	$18.49_{\pm 0.99}$	–
	Security Only SFT	$48.10_{\pm 1.49}$	+14.37	$27.94_{\pm 2.15}$	+9.45
	SECUREVIBE$_{rl}$	$48.33_{\pm 2.48}$	+14.60	$28.25_{\pm 1.20}$	+9.76
	Simple Mixture SFT	$43.41_{\pm 1.46}$	+9.68	$24.05_{\pm 1.91}$	+5.56
	SECUREVIBE$_{base}$	$47.70_{\pm 2.65}$	+13.97	$32.22_{\pm 2.00}$	+13.73
	SECUREVIBE$_{rl}$	$50.95_{\pm 0.82}$	+17.22	$33.89_{\pm 0.69}$	+15.40
	SECUREVIBE$_{hg}$	$46.82_{\pm 0.60}$	+13.09	$32.94_{\pm 0.50}$	+14.45

Explicit security supervision matters more than simply adding secure coding trajectories. SECUREVIBE$_{base}$ delivers the strongest SFT security profile on both benchmarks in Table 1. Compared with the similarly sized Simple Mixture, it achieves higher SecPass on PatchEval-Gen and AutoBaxBench, demonstrating the value of explicit planning and testing-synthesis supervision. Instead, Security Only improves functional task completion but does not reliably improve secure solutions. This indicates that with trajectories that pass both functionality and security, the agent only learns how to mimic the functionality implementation behaviors, but still lacks security awareness.

SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$ further strengthens security capabilities. We further train SECUREVIBE$_{rl}$ on Security Only SFT and Security Suite SFT, and apply SECUREVIBE$_{hg}$ to the better Security Suite SFT checkpoint to compare performance between SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$. As shown in Table 1, compared to SECUREVIBE$_{base}$, the post-training SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$ improve security capabilities in all 3 scenarios on 2 benchmarks. However, SECUREVIBE$_{hg}$ hurts FuncPass on both benchmarks. We find this is because providing the model with security hints sometimes makes it focus too much on security at the cost of functional correctness, which aligns with the finding in SusVibes (Zhao et al., 2026b). This also indicates that simply adding security hints in the prompt cannot reliably improve the security

Table 2: Security Generalization. SFT is SECUREVIBE$_{base}$. Bold indicates the best mean for each benchmark and metric among Qwen 3.5 35B variants.

		Security Coding				*Generic Coding*
		SusVibes (186)		BaxBench (392)		SWE-bench Verified (500)
Model	Size	FuncPass	SecPass	FuncPass	SecPass	Pass@1
GPT 5.6 Sol	–	$85.83_{\pm 1.63}$	$22.40_{\pm 1.56}$	$57.91_{\pm 0.44}$	$45.32_{\pm 0.39}$	$96.20^{*}_{\pm 0.86}$
MiniMax M2.7	230B	$53.23_{\pm 2.16}$	$13.98_{\pm 1.73}$	$58.16_{\pm 2.21}$	$33.42_{\pm 1.32}$	$73.80^{*}_{\pm 1.97}$
Qwen 3.5 35B	35B	$28.14_{\pm 0.82}$	$10.57_{\pm 0.62}$	$32.91_{\pm 2.87}$	$19.71_{\pm 1.15}$	$60.90_{\pm 1.50}$
SECUREVIBE$_{base}$	35B	$41.76_{\pm 2.54}$	$13.62_{\pm 3.10}$	$36.48_{\pm 1.56}$	$25.34_{\pm 1.31}$	$\mathbf{65.00}_{\pm 1.20}$
SECUREVIBE$_{rl}$	35B	$41.76_{\pm 1.73}$	$13.26_{\pm 0.82}$	$\mathbf{41.07}_{\pm 0.77}$	$\mathbf{26.62}_{\pm 1.88}$	$63.40_{\pm 0.20}$
SECUREVIBE$_{hg}$	35B	$39.96_{\pm 3.24}$	$\mathbf{14.16}_{\pm 1.35}$	$38.26_{\pm 2.88}$	$23.64_{\pm 0.64}$	$64.70_{\pm 0.70}$

* Results from the Vals AI SWE-bench leaderboard: https://www.vals.ai/benchmarks/swebench.

performance without hurting functionality. This also highlights the necessity of learning security capabilities through a suitable training recipe.

4.2 SECURE CODING GENERALIZATION

To verify the generalization of our security training, we evaluate on two more security benchmarks, BaxBench and SusVibes [1], and one generic coding benchmark, SWE-bench Verified. We further compare SECUREVIBE performance on GPT 5.6 Sol and MiniMax M2.7, which are the models from which our SFT and hints are distilled.

Security capabilities are generalized to new security benchmarks. Table 2 shows consistent gains on both SusVibes and BaxBench for all trained configurations. This consistency suggests that the models learn security awareness and generalizable security capabilities. SECUREVIBE$_{hg}$ achieves the strongest transfer on SusVibes, while SECUREVIBE$_{rl}$ performs best on BaxBench.

We further examine the security gap, defined as FuncPass- SecPass, which measures the performance gap between security and functionality capabilities. Ideally, this gap should be 0, meaning that every functionality-correct solution should also be secure. Our SECUREVIBE shows a smaller gap than the more powerful GPT 5.6 Sol and MiniMax M2.7 on SusVibes and outperforms MiniMax M2.7 on BaxBench, showing that our training recipe efficiently mitigates the security gap. Besides, SECUREVIBE$_{hg}$ shows better performance on SusVibes, which aligns with its superior performance on PatchEval-Gen. In these two tasks, SECUREVIBE$_{base}$ performs worse in security, making it more difficult to gain positive executable signals during reinforcement learning. However, benefiting from hint-based supervision, SECUREVIBE$_{hg}$ can still gain effective training signals for security.

Security training preserves general coding ability on SWE-bench Verified. Table 2 shows that security training also improves SWE-bench Verified performance, showing that security planning and testing strengthen general repository-level coding capabilities. Trajectory analysis further shows that successful solutions more often modify implementation code without changing tests or documentation (37 of 38 discordant pairs; 97.4%; $p < 0.001$), confine the patch to a single file (42 of 45; 93.3%; $p < 0.001$), and run targeted validation after editing (11 of 12; 91.7%; $p = 0.006$). These results show that SECUREVIBE preserves the focused editing and direct validation behaviors required for general software engineering.

5 ANALYSIS

5.1 HOW DOES TRAINING CHANGE SECURITY BEHAVIORS?

We investigate the behavioral changes associated with our training recipe. As in Section § 3.1, we examine security planning, coding, and testing in agent trajectories.

[1] We find that the agent frequently searches online for the original patches when implementing features in SusVibes, so we explicitly forbid online retrieval via git during evaluation.

SECUREVIBE$_{base}$ strengthens Security Planning, Coding, and Testing, and SECUREVIBE$_{hg}$ more consistently reinforces security behaviors Figure 4 shows Security Suite consistently elicits the greatest improvement in planning, coding, and testing on both benchmarks, demonstrating that structured supervision activates security behaviors more effectively than coding trajectories alone. Figure 5 compares each post-training method with Security Suite SFT. SECUREVIBE$_{hg}$ broadly reinforces planning, coding, and testing on both benchmarks, with particularly strong planning and coding gains on PatchEval-Gen. This verifies that the security supervision from the security hints is more fine-grained than outcome-based SECUREVIBE$_{rl}$, making it a more direct way to further enhance the security behaviors. Such a trend is hidden from SecPass in Table 1 because of its inferior performance on functionality correctness.

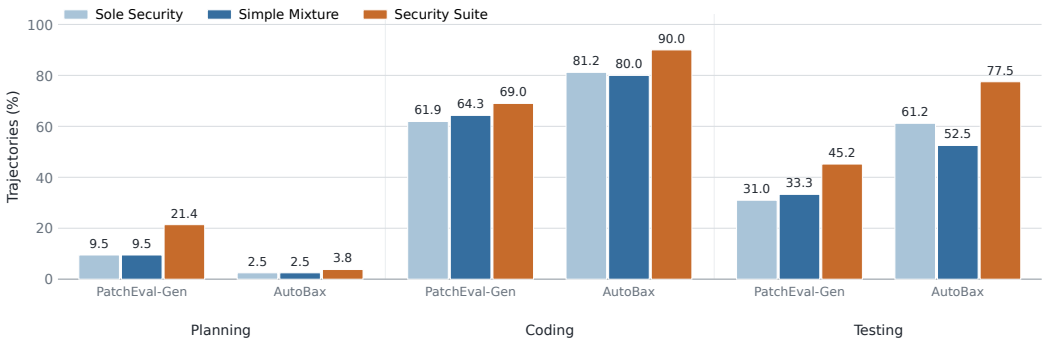

Figure 4: Percentage of trajectories exhibiting security planning, coding, and testing under different SFT recipes. Security Suite has the highest prevalence of all three behaviors on both benchmarks.

5.2 WHY DO SECUREVIBE$_{hg}$ AND SECUREVIBE$_{rl}$ EXPERT IN DIFFERENT BENCHMARKS ?

To further investigate how different post-training methods affect the learning of security capabilities, we compare their performance on different CWEs. The CWE results on BaxBench in Figure 6 show that SECUREVIBE$_{rl}$ shows larger gains for *exception handling, code injection, cross-site scripting, SQL injection, and access control*, while SECUREVIBE$_{hg}$ is stronger for *file upload and path traversal*.

To understand these differences, we examine their training process. In the analyzed rollout data, SECUREVIBE$_{rl}$ retains more positive rollouts than SECUREVIBE$_{hg}$ for *exception handling, SQL injection, cross-site scripting, command injection, and access control*. We further examine the security hints used for SECUREVIBE$_{hg}$. Compared with the other CWEs, hints for CWE-22 path-traversal can be more detailed and explicit (*canonical-path containment, parent-directory traversal, symlinks, filenames and extensions, and confining filesystem operations to an allowed root*). Such guidance explains SECUREVIBE$_{hg}$'s CWE-22 improvement on CWE-22. Moreover, CWE-434 gains no successful rollouts during training for SECUREVIBE$_{rl}$, but this vulnerability is similar to CWE-22, which explains SECUREVIBE$_{hg}$'s better performance on it.

Further, when looking back at SECUREVIBE$_{rl}$'s better security performance on AutoBaxBench (and similar BaxBench) and its less ideal performance on PatchEval-Gen (and SusVibes), we find that this is also because of the density of the outcome reward: approximately 30% of SECUREVIBE$_{rl}$ rollout trajectories on AutoBaxBench receive a reward of 1 by passing both the functional and security tests, compared with only 15% on PatchEval-Gen. In contrast, SECUREVIBE$_{hg}$ relies more on teacher supervision, making it possible to still improve with a sparse outcome signal.

5.3 DO SECURITY CAPABILITIES TRANSFER TO UNSEEN CWES?

We define an unseen CWE as an exact CWE ID absent from both training datasets. The SusVibes subset contains 48 unseen IDs across 78 instances: 71 contain only unseen IDs, and seven contain both seen and unseen IDs. The BaxBench subset contains three unseen IDs, CWE-117, CWE-400, and CWE-434, across 140 disjoint instances.

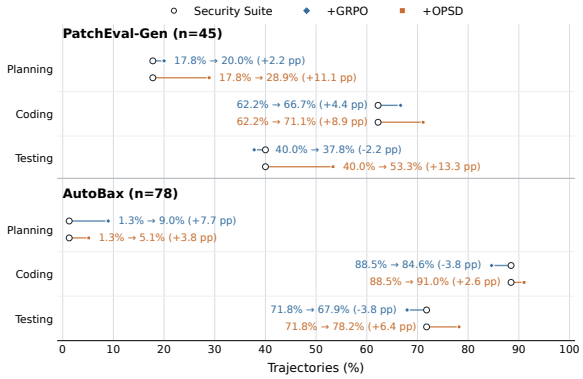

Figure 5: Security-behavior changes relative to Security Suite SFT. Labels show percentage-point changes. SECUREVIBE$_{hg}$ broadly increases all three behaviors, while SECUREVIBE$_{rl}$ concentrates its largest gain in security planning on AutoBaxBench.

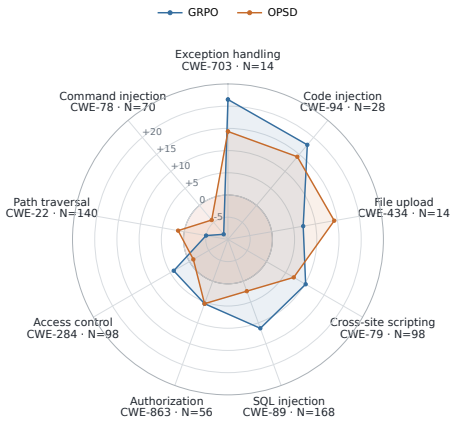

Figure 6: Per-CWE changes in SecPass on BaxBench relative to Security Suite SFT, in percentage points. N denotes the number of instances in each category; categories may overlap.

Table 3: Generalization performance on instances containing unseen CWEs. Parenthesized values denote percentage-point changes from the Qwen 3.5 35B baseline. The best score in each row is bold.

Dataset	# Instances	Metric	Qwen 3.5 35B	SECUREVIBE$_{base}$	SECUREVIBE$_{rl}$	SECUREVIBE$_{hg}$
SusVibes	78	FuncPass	20.51	30.77 (+10.26)	33.33 (+12.82)	**38.46** (+17.95)
		SecPass	7.69	12.82 (+5.13)	15.38 (+7.69)	**19.23** (+11.54)
BaxBench	140	FuncPass	31.43	33.57 (+2.14)	39.29 (+7.86)	**40.00** (+8.57)
		SecPass	10.71	13.57 (+2.86)	**17.86** (+7.14)	13.57 (+2.86)

Table 3 shows that the learned security capabilities transfer to vulnerability categories absent during training. SECUREVIBE$_{hg}$ provides the strongest transfer on SusVibes, while SECUREVIBE$_{rl}$ is more effective for secure implementation on BaxBench, reinforcing their complementary strengths across task formats. On SusVibes, instance-level paired tests confirm that the functional gains from SECUREVIBE$_{hg}$ ($p = 0.0013$) and SECUREVIBE$_{rl}$ ($p = 0.0414$) are statistically significant. The security gain from SECUREVIBE$_{hg}$ is also significant ($p = 0.0117$). This suggests that hint-guided distillation transfers security knowledge more reliably to unseen vulnerability types in repository-level feature request tasks.

6 CONCLUSION

We study how coding agents can produce secure solutions beyond functional correctness. Our trajectory analysis reveals that the insecure agent is less likely to perform planning, coding, and testing for implicit security requirements This motivates SECUREVIBE, a training recipe that combines structured security-focused SFT with SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$. Across evaluation of two types of repo-level coding tasks on 4 benchmarks, SECUREVIBE improves functional and secure task completion over the base model. The gains extend to instances with unseen CWE categories while preserving and further improving general coding performance such as SWE-bench Verified.

Our findings highlight two takeaways: (i) how security supervision is structured matters more than simply adding secure coding trajectories: explicit planning and testing tasks more effectively elicit security capabilities; and (ii) verifiable security outcomes do not always provide sufficient learning signals. SECUREVIBE$_{rl}$ with positive-reward dynamic filtering is effective when successful outcomes are sufficiently frequent, while SECUREVIBE$_{hg}$ provides detailed teacher hints when agents rarely succeed. Together, these findings suggest that improving secure vibe coding requires teaching agents how to uncover hidden risks and how to make progress before they can reliably earn positive rewards.

AI USE STATEMENT

In this work, we used generative AI tools to polish the writing, figures and tables. We did not use generative AI tools to formulate mathematical claims, develop or write proofs, or translate research materials; the remaining required-disclosure tasks were not applicable to this work. The authors reviewed all AI-assisted work. We validated generated trajectories and code using the corresponding functional and security tests. Generative AI did not determine the conclusions of this work. The authors take responsibility for the final content of the paper, including all text, claims, analyses, and artifacts produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

We provide the code and data needed to reproduce our training and evaluation pipeline in the anonymous repository at https://anonymous.4open.science/r/safevibe-anonymous-87FD/ (also attached as supplementary material). The training recipe and objectives are described in § 3, while the benchmark splits, training configurations, metrics, and evaluation settings are detailed in § 4. The appendix provides additional information on security-behavior annotation, analysis, supervised data construction, and more training parameters.

REFERENCES

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. Swe-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026.

Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. Secureagentbench: Benchmarking secure code generation under realistic vulnerability scenarios, 2025. URL https://arxiv.org/abs/2509.22097.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. Envbench: A benchmark for automated environment setup. In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=MgTzMaYHvG.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Minseon Kim, Zhengyan Shi, Emiliano Penaloza, Christopher Cui, Roger Creus Castanyer, Maryam Hashemzadeh, Isadora White, Jonathan Light, Jeonghye Kim, Matheus Pereira, et al. Frognano: Training a 4b coding agent via online task synthesis. *arXiv preprint arXiv:2609.07925*, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. *arXiv preprint arXiv:2503.18470*, 3:1, 2025.

Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, et al. Purpcode: Reasoning for safer code generation. *Advances in Neural Information Processing Systems*, 38:55146–55200, 2026.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *CoRR*, 2025.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37: 81857–81887, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Forty-second International Conference on Machine Learning*, 2025.

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. *Communications of the ACM*, 68(2):96–105, 2025.

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

Jiazheng Quan, Xiaodong Li, Bin Wang, Guo An, Like Liu, Degen Huang, Lin Liu, and Chengbin Hou. Learning to generate secure code via token-level rewards. *arXiv preprint arXiv:2602.23407*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. Sallm: Security assessment of generated code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pp. 54–65, 2024.

Mingyang Song and Mao Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

Maximilian Thang, Lichao Wu, Sasha Behrouzi, Mohamadreza Rostami, Jona te Lintelo, Stjepan Picek, and Ahmad-Reza Sadeghi. GoodVibe: Security-by-vibe for LLM-based code generation. *arXiv preprint arXiv:2602.10778*, 2026.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. Baxbench: Can llms generate correct and secure backends? *arXiv preprint arXiv:2502.11844*, 2025.

Tobias von Arx, Niels Mündler, Mark Vero, Maximilian Baader, and Martin Vechev. Autobaxbuilder: Bootstrapping code security benchmarking. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=J5KrubuRAJ.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a. URL https://openreview.net/forum?id=ULblO61XZ0.

Zichao Wei, Jun Zeng, Ming Wen, Zeliang Yu, Kai Cheng, Yiding Zhu, Jingyi Guo, Shiqi Zhou, Le Yin, Xiaodong Su, et al. Patcheval: A new benchmark for evaluating llms on patching real-world vulnerabilities. *arXiv preprint arXiv:2511.11019*, 2025b.

Tianyi Wu, Mingzhe Du, Yue Liu, Chengran Yang, Terry Yue Zhuo, Jiaheng Zhang, and See-Kiong Ng. SecCoderX: Secure code generation via online reinforcement learning with vulnerability reward model. *arXiv preprint arXiv:2602.07422*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 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. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024a.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b. URL https://arxiv.org/abs/2405.15793.

Yu Yang, Yuzhou Nie, Zhun Wang, Yuheng Tang, Wenbo Guo, Bo Li, and Dawn Song. Seccodeplt: A unified platform for evaluating the security of code genai. *arXiv preprint arXiv:2410.11096*, 2024c.

Jingheng Ye, Huiqi Zou, Simon Yu, and Weiyan Shi. Coding with" enemy": Can human developers detect ai agent sabotage? *arXiv preprint arXiv:2606.05647*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. Sealign: Alignment training for software engineering agent. *CoRR*, 2025.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. In *Forty-third International Conference on Machine Learning*, 2026a. URL https://openreview.net/forum?id=Jpxfof0EaS.

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 *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=qG8g00zRZa.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

A Security Behavior Annotation

We conduct two kinds of security behavior annotations: generic security behaviors include all kinds of security attempts, and target-aligned security behaviors only care about the security attempts that are related to the ground-truth vulnerability.

Generic Security Behavior Annotation. We use GPT-5.6-Sol to identify security behaviors in assistant messages, commands, visible tool outputs, and submitted patches. Each dimension is independently labeled `present`, `absent`, or `uncertain`, with positive labels requiring exact evidence excerpts and source indices:

- **Planning:** identifies a task-specific security risk and an intended inspection, mitigation, or validation. Generic security advice and risk mentions without an intended action are excluded.
- **Coding:** implements or restores a concrete protective control in production code, supported by an edit command and an added line in the submitted patch. Test-only edits, comments, imports, and read-only inspection are excluded.
- **Testing:** authors a security-specific check, executes an existing identified security check, or invokes a deliberate adversarial probe. Evidence must identify the security target and the relevant command; execution claims also require visible results. Promises to test and generic test-suite invocations without a security target are excluded.

Target-Aligned Security Annotation. A separate GPT-5.6-Sol pass compares recorded events with SusVibes reference security patches, test patches, and CWE/CVE metadata. Model identity, final benchmark outcomes, and prior alignment and quality labels are withheld. Events are labeled `direct` if they address the same security property, `related` if they concern the same component or threat family but a different property, `unrelated`, or `uncertain`. A dimension is target-aligned if at least one event is `direct`; partial or unsuccessful protections can qualify. Alignment records include event and reference quotations and a property-comparison rationale. Mechanical checks verify fields and quoted evidence, not semantic accuracy. The alignment pass cannot recover events omitted during initial annotation.

Evaluation. We measure associations between binary behavior indicators and security outcomes using Pearson's phi. For matched tasks, we compute Pearson correlations between GPT-minus-Qwen behavior differences and security-outcome differences. Uncertainty is estimated from 5,000 project-level bootstrap resamples, preserving within-project dependencies and model pairs. Reported percentile 95% intervals are pointwise, not multiplicity-adjusted. Resamples with undefined correlations are excluded and counted in the diagnostics.

B Generic Security Behavior

We compare generic security-behavior presence with target-aligned presence in 166 functionally correct GPT-5.6-Sol trajectories on SusVibes.

Table 4: Associations between security-behavior presence and security passing. Entries show Pearson's phi with pointwise project-bootstrap 95% intervals.

Behavior	Generic	n	Target-aligned	n
Planning	-0.055 $[-0.195, +0.124]$	166	$+0.285$ $[+0.111, +0.443]$	166
Coding	-0.116 $[-0.205, +0.025]$	166	$+0.469$ $[+0.297, +0.612]$	166
Testing	-0.194 $[-0.298, -0.065]$	165	$+0.199$ $[+0.010, +0.359]$	166

Table 4 shows positive associations for all behaviors aligned with the target, with intervals above zero. Generic planning and coding have zero-day intervals, while generic testing has a negative association. Generic testing excludes one uncertain annotation, so its sample differs slightly from the target-aligned analysis. These results suggest that relevance to the evaluated security requirement matters more than security-oriented activity alone.

Table 5: Distribution of benchmark instances across programming languages. A dash indicates that a language is not included in the benchmark.

Language	PatchEval-Gen	AutoBaxBench	BaxBench	SusVibes
JavaScript	75	160	112	–
Python	44	160	112	186
Go	81	120	84	–
Ruby	–	40	28	–
Rust	–	40	28	–
PHP	–	40	28	–
Total	**200**	**560**	**392**	**186**

Table 6: Benchmark instances categorized by CWE theme. A single instance may contain multiple CWEs and can therefore be counted under multiple themes.

CWE Theme	PatchEval-Gen	AutoBaxBench	BaxBench	SusVibes
Injection	53	294	252	14
XSS	4	182	98	24
Path Traversal	40	154	140	15
Access Control	45	252	140	13
Auth & Credentials	11	182	70	12
Input Validation	12	280	28	7
Resource/DoS	0	0	112	23
Error/Exception	0	560	392	10
Request Forgery	27	0	0	20
Sensitive Info	8	0	14	18
Crypto	2	0	0	9

Generic behavior includes aligned activities and therefore does not isolate irrelevant behavior. Its negative testing association does not establish that irrelevant testing harms security. These observational results may reflect task difficulty or other confounders; the intervals neither test differences between definitions nor account for annotation error or selection bias.

C Benchmark Details

In Table 5 and Table 6, we list the programming-language and CWE-theme distributions of the four benchmarks.

PatchEval-Gen. PatchEval (Wei et al., 2025b) is a benchmark for evaluating agents' capabilities to patch real-world vulnerabilities. It includes 1,000 instances, each containing a CVE description and vulnerable and secure code patches. Among them, 230 instances contain executable Docker environments. The original task setting asks the agent to fix the vulnerable code and uses security test cases to determine whether the vulnerability has been fixed. We adapt the SusVibes curation pipeline (Zhao et al., 2026b) to reproduce these tasks for secure code generation. Specifically, we start from the vulnerable code, mask the corresponding feature implementation, and create a feature request based on it. We mine functional tests from the original repository and use the original security tests to measure the security pass rate. To avoid overlap with SusVibes, we remove instances that share the same CVE ID. Finally, we obtain 200 instances with 1.52 CWEs per instance on average, which we call PatchEval-Gen.

AutoBaxBench. AutoBaxBench (von Arx et al., 2026) is created using AutoBaxBuilder, an automated framework for generating code-security benchmarks for backend tasks. AutoBaxBench contains 40 scenarios spanning 14 frameworks and languages. Each scenario has approximately 3.4 CWEs.

BaxBench. BaxBench (Vero et al., 2025) consists of security-critical backend coding tasks. It contains 28 coding scenarios with 14 popular backend development frameworks. Note that there is no overlap with the 40 AutoBaxBench scenarios.

SusVibes. SusVibes (Zhao et al., 2026b) is a real-world security coding benchmark with 186 tasks. Each task requires the coding agent to implement a security-sensitive feature. SusVibes contains 76 CWEs; each instance has at least one CWE, and 12 instances (6.5%) contain two CWEs.

We use PatchEval-Gen and AutoBaxBench as the training sets and BaxBench and SusVibes as the cross-domain test sets. We use 100 PatchEval-Gen and 140 AutoBaxBench instances to build the supervised training data and the remaining 100 and 420 instances for SECUREVIBE$_{rl}$ and SECUREVIBE$_{hg}$ post-training.

D SUPERVISED DATA CURATION

As mentioned in Section 3.2, we build our supervised security suite from four tasks: Functionality-Focused Coding, Security-Focused Coding, Security Planning, and Security Test. We use the open-weight model MiniMax M2.7 with 230B parameters and 10B active parameters. We sample 20 times per instance on 100 PatchEval-Gen and 140 AutoBaxBench instances to build the SFT data.

In Functionality-Focused Coding, we use the original test description as the input and keep trajectories that are functionally correct and secure. The agent is asked to implement it with only generic security guidance (*Make sure to follow best security practices and avoid common vulnerabilities when resolving this issue*). We keep trajectories that pass both functionality and security tests.

In Security-Focused Coding, we provide additional security information for each instance, including the target CWE or CVE and its description. The agent is asked to implement the feature to avoid these vulnerabilities (*In addition to satisfying the functionality above, make sure your implementation is secure with respect to these CWE categories*). Similarly, we only keep joint pass trajectories.

In Security Planning, we provide the original test description and a list of potential CWEs and ask the agent to identify the relevant ones (*Identify the potential vulnerability or vulnerabilities of this feature. Include only the most specific CWE categories that are directly caused by the feature. Prefer a small primary set over broad coverage.*) We keep the trajectories that identify the correct CWE compared with the ground truth.

In Security Test, we provide the ground-truth CWE and ask the agent to create the test cases based on them (*Generate one minimal, apply-ready security unit test patch that verifies the most important security property.*). We evaluate the ground-truth secure and vulnerable code on these generated tests. If the generated test can be executed and distinguish these two code versions, we keep this trajectory.

Finally, we collect successful trajectories from all tasks and create 3 different recipes for SFT training.

E TRAINING PARAMETERS

We use Slime (Zhu et al., 2025) for SFT fine-tuning and RL training. We fine-tune Qwen3.5-35B-A3B using token-level cross-entropy on assistant responses, including reasoning content. We use a maximum sequence length of 64K tokens.

Reinforcement learning. The policy generates multi-turn coding trajectories, and submitted solutions are evaluated using functional and security tests. Let $F, S \in \{0, 1\}$ indicate whether the respective checks pass. For tasks with both checks, the terminal reward is

$$R(F,S) = \begin{cases} 1, & F=1,\ S=1, \\ 0.5, & F=1,\ S=0, \\ -0.5, & F=0, \end{cases} \tag{3}$$

It provides partial credit for functional correctness while penalizing functional regressions even when security tests pass. Invalid submissions receive -1, and submissions that cannot be applied receive -0.75. Unresolved grading failures receive 0. Optimization uses Adam with a peak learning rate of 10^{-5}, a 10% warmup, cosine decay to 10^{-6}, and weight decay of 0.1.

We optimize the policy with SECUREVIBE$_{rl}$, sampling 4 trajectories per prompt and 8 prompts per update, for a batch size of 16. Rollouts use temperature 1.0 and an 8,192-token response limit. Dynamic sampling retains groups with no aborted trajectories, nonzero reward variance, and at least one positive reward. We use a learning rate of 10^{-6} with cosine decay, lower and upper clipping parameters of 0.2 and 0.28, and an entropy coefficient of 10^{-4}.

Hint Generation. We define four cumulative hint-exposure levels, where each level keeps the previous one and adds more detail. All hints are generated by GPT 5.6 Sol. We provide the task description, the metadata with CWE and CVE information, and the security test cases to GPT 5.6 Sol and ask it to provide hints with different levels. L1 uses the expected vulnerability categories from the dataset metadata. L2 adds the security boundary and the property that must hold. L3 adds a suggested verification workflow. L4 adds oracle-level detail, such as concrete trigger inputs and the expected secure behavior.

Two training benchmarks differ in how these hints are applied. In PatchEval-Gen, the hints follow a generic security reminder, so L1 is the first time the CWEs are revealed, and the later levels describe regression tests for the repair. In AutoBaxBench, the hints follow a prompt that already lists the scenario's CWEs, so L1 only narrows attention to the exploited ones. The later levels are organized per vulnerable boundary and describe exploit workflows and attack payloads. We find that L1 barely improves security, which agrees with the SusVibes observation (Zhao et al., 2026b). In contrast, L4 improves security more significantly. This suggests that the bottleneck is not knowing the vulnerability category but turning it into a concrete threat against the code being written. We therefore use L4 hints for OPSD training in our main experiments.

On-policy distillation. SECUREVIBE$_{hg}$ combines the task-reward objective with token-level teacher supervision on student-generated trajectories. The student receives the task prompt, while the teacher additionally receives instance-specific security hints. Teacher probabilities are aligned to the student's generated tokens, providing privileged security guidance during training without exposing these hints to the student. We combine SECUREVIBE$_{rl}$ with a skew Jensen–Shannon distillation surrogate, using a distillation coefficient of 1.0 and a teacher mixture weight of 0.5. The task reward follows the definition above. We sample 8 trajectories per prompt and 4 prompts per update, and use a constant learning rate of 10^{-6}. The sampling filter excludes aborted groups but does not require reward variation, allowing teacher supervision to contribute even when all trajectories receive the same task reward.