EVOSECCODER: ADVERSARIAL CO-EVOLUTION FOR SECURE CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models increasingly excel at code generation, but still generate vulnerable code. To generate more secure code, models need to learn to follow secure examples and avoid insecure patterns without losing functionality. Models can learn from secure and insecure examples drawn from a fixed dataset or generated by the model itself. However, fixed examples become less useful as the model learns, while the model's own functional insecure examples become rarer as its code becomes more secure. Both sources therefore provide less training signal as the model improves. We present EVOSECCODER, an adversarial co-evolution framework that jointly evolves secure and insecure models from the same base model while preserving functionality. The two models co-evolve by supplying adversarial samples to each other. In each evolution round, both models solve the same tasks, and executable tests verify their programs. Functional but insecure programs challenge the secure model, while secure programs guide the insecure model toward new insecure alternatives. Across five open models, with training rounds selected on held-out validation CWEs, EVOSECCODER improves the mean SecCodeBench correct-and-secure rate on the remaining test CWEs from 17.4% to 23.3%, while mean EvalPlus functional correctness changes from 65.6% to 64.9%. Removing or freezing the insecure model reduces the mean correct-and-secure rate to 21.1% or 21.8%, respectively; these ablations support the value of a dedicated, co-evolving insecure model beyond additional sampling alone.

1 INTRODUCTION

Code language models can generate programs that appear correct but contain security vulnerabilities (Pearce et al., 2022). Several benchmarks therefore evaluate generated code for security as well as

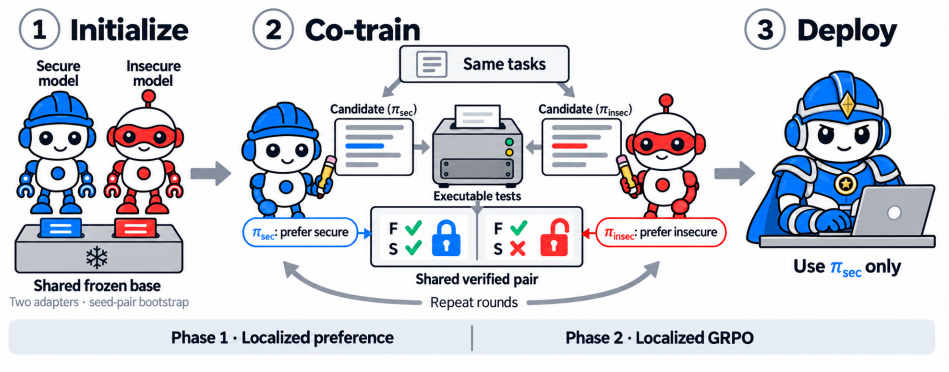

Figure 1: EVOSECCODER at a glance. (1) $\pi_{\mathbf{sec}}$ and $\pi_{\mathbf{insec}}$ are two adapters over one frozen base model, bootstrapped on seed pairs. (2) Both policies generate code for the same tasks. Executable tests guide the construction of shared pairs, from which the policies learn opposite preferences. Each policy receives localized GRPO updates on its own fresh rollouts. (3) Only $\pi_{\mathbf{sec}}$ is used for coding.

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.

functionality (Peng et al., 2025; SecCodeBench Team, 2025; Nie et al., 2025). On such benchmarks, a substantial portion of the code that passes functional tests still fails security tests (Peng et al., 2025).

Several methods improve code security through supervised or preference learning on secure and insecure examples. SVEN and SafeCoder train on curated vulnerability-fixing examples (He & Vechev, 2023; He et al., 2024), and DiSCo distills such pairs from frontier models and focuses learning on the tokens where the two programs differ (Hasan et al., 2025). ProSec and HexaCoder instead synthesize security-focused training data (Xu et al., 2025; Hajipour et al., 2024). Online RL methods instead learn from rewards on the model's own samples (Yao et al., 2025; Liu et al., 2025; Wu et al., 2026). TSP applies self-play to secure code generation by sampling insecure alternatives from the model itself at annotated risk nodes (Chen et al., 2026). In the pair-based and self-play approaches, the insecure code used for learning is either prepared outside the training loop or produced by the model being trained. Once collected, a fixed preference dataset does not refresh its examples as the policy is updated. Self-generated insecure code, in turn, may become less challenging as the model improves, as the TSP authors note.

Motivated by these limitations, we propose EvoSecCoder, an adversarial dual-model self-training framework with two adapters over the same frozen code model. Both policies contribute generated code to shared preference pairs and learn opposite preferences from them: $\pi_{\mathbf{sec}}$ favors secure code, while $\pi_{\mathbf{insec}}$ favors functional but insecure code. This separate policy supplies insecure code beyond $\pi_{\mathbf{sec}}$'s own generations. We update $\pi_{\mathbf{insec}}$ alongside $\pi_{\mathbf{sec}}$ to refresh this source during training, and prefer its qualifying programs with high likelihood under the current $\pi_{\mathbf{sec}}$.

Both policies receive the same programming prompts, and executable tests decide which of their outputs are used for learning. A generated program can enter a pair only if it passes the functional tests and receives the required security verdict. Because code that omits required behavior may pass a security test without solving the task (Peng et al., 2025), this requirement keeps functional failure from being mistaken for insecurity. Each round first trains both policies on shared pairs with opposite preferences, using only the tokens where the paired code differs. The updated policies then generate code that is tested to build the next round's pairs. Finally, each policy learns from its own fresh rollouts through online RL with test-based rewards favoring opposite security outcomes. Figure 1 summarizes this process.

We evaluate whether $\pi_{\mathbf{sec}}$ improves joint correctness and security over the base model and existing methods while preserving functionality. We assess the contribution of $\pi_{\mathbf{insec}}$ by removing it or freezing its parameters. We choose training rounds on a validation split of SecCodeBench whose vulnerability types (CWEs) do not appear in the reported test split. Across five open code models, the selected checkpoints of EvoSecCoder raise the mean test correct-and-secure rate from 17.4% to 23.3%, above all four published baselines (17.4% to 20.0%), while mean EvalPlus functional correctness changes from 65.6% to 64.9%. The variants that remove or freeze $\pi_{\mathbf{insec}}$ reach mean correct-and-secure rates of 21.1% and 21.8%, respectively.

Our contributions are as follows.

- We propose EvoSecCoder, an adversarial dual-model self-training framework in which an insecure policy, updated alongside the secure policy on the same coding prompts, supplies code for the rejected side of the secure policy's preference pairs only when it passes the functional tests and fails the security tests.
- We design a round-based training procedure that alternates between preference learning on shared pairs and online RL on each policy's own code, refreshing the pair pool between the preference and online phases of each round.
- We evaluate EvoSecCoder on five open code models against published baselines and component variants, measuring joint correctness and security, functional correctness, and generalization to a language and CWE categories absent from replay data.

2 Related Work

2.1 Joint Functionality and Security

CWEval (Peng et al., 2025) evaluates the functionality and security of generated code on the same programs. SecCodeBench (SecCodeBench Team, 2025) measures both properties on project-

based test cases, and SeCodePLT (Nie et al., 2025) uses dynamic tests to evaluate code generation, vulnerability detection, and patching. EvalPlus (Liu et al., 2023) strengthens functional tests so that more functional errors are caught. CWEval also shows that evaluating the two properties separately can miss functionality losses on security-related tasks. We therefore evaluate security and functionality together on the same code.

2.2 Learning from Prepared Secure and Insecure Code

To improve the security of generated code, a common approach is to train on secure and insecure code prepared in advance, using supervised or preference learning. The data come from vulnerability-fixing commits, as in SVEN (He & Vechev, 2023) and SafeCoder (He et al., 2024), from frontier models, as in DiSCo (Hasan et al., 2025) and HexaCoder (Hajipour et al., 2024), or from the target model itself, as in ProSec (Xu et al., 2025). These methods do not maintain a separately trained policy that supplies new insecure code for the same tasks during training.

2.3 Learning from Code Generated during Training

Another approach is to learn from code written during training. REAL (Yao et al., 2025), PurpCode (Liu et al., 2025), SecCoderX (Wu et al., 2026), and SRCode (Quan et al., 2026) apply online RL to the model's own samples, with rewards from static analyzers, unit tests, learned vulnerability models, or token-level security signals. Self-play methods such as SPIN (Chen et al., 2024) and TSP (Chen et al., 2026) contrast reference responses with the model's own outputs. TSP samples insecure alternatives from the policy itself.

These observations motivate our design. We train a separate insecure policy alongside the secure policy on the same prompts to generate new insecure code as training progresses. §3 describes the procedure.

3 Design

EvoSecCoder trains two policies toward opposite security goals. $\pi_{\mathbf{sec}}$ learns to generate secure code and is the policy returned by the method. $\pi_{\mathbf{insec}}$ learns to generate functional but insecure code that is used to train $\pi_{\mathbf{sec}}$. Both policies are LoRA adapters (Hu et al., 2022) over the same frozen code model, and both receive the same task prompts without role-specific instructions. Executable functionality and security tests label the code that the policies generate, and the labels decide which code is used for learning.

Figure 2 summarizes the training process. We first bootstrap both policies on existing pairs of secure and insecure code (§3.1). Training then proceeds in rounds. Let W_r denote the task window for round r, a batch of replay tasks sampled for that round. Let P_r denote the pair pool, the set of code pairs used for preference learning in round r. Before the first round ($r = 0$), both policies generate rollouts for an initial task window, and we turn them into P_0 (§3.2). In each round, Phase 1 trains the two policies on shared data. Both learn from the same pair pool P_r, built from the rollouts of both policies, but in opposite directions, so that $\pi_{\mathbf{sec}}$ learns to prefer the secure code and $\pi_{\mathbf{insec}}$ the functional but insecure code of each pair (§3.3). After Phase 1, both policies generate rollouts on W_r to construct P_{r+1} for the next round and select tasks for Phase 2. Phase 2 then trains each policy on its own data. Each policy learns from the code it newly generates for its selected tasks, using localized GRPO with test-based rewards that again favor opposite outcomes for the two policies, and the updated adapters start the next round (§3.4).

3.1 Dual-Model Initialization

Before the first round, we train two LoRA adapters on the frozen code model into their opposite roles. The training data are pairs of secure and insecure implementations of the same task from SafeCoder (He et al., 2024) and SeCodePLT (Nie et al., 2025). We use these pairs as provided by the data sources and do not run our executable tests on them, so unlike the generated code in later rounds, bootstrap pairs are not filtered by the functional tests.

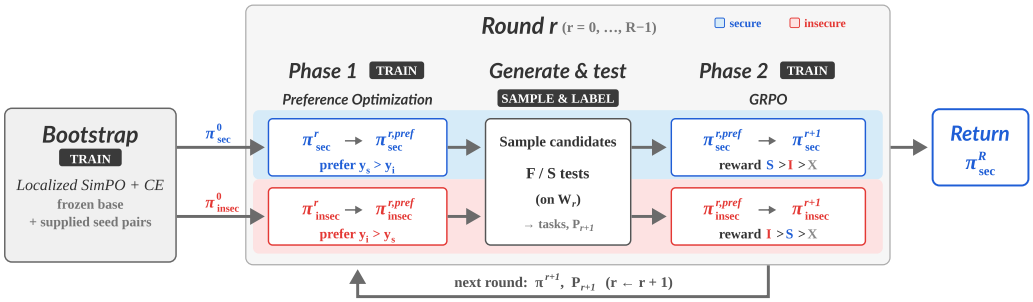

Figure 2: Training overview of EvoSecCoder. After bootstrap, the secure (blue) and insecure (red) policies alternate between learning opposite preferences on shared code pairs (Phase 1) and localized GRPO on their own fresh rollouts (Phase 2). Between phases, functionality/security tests (F/S) select tasks and build pairs for the next round. S/I/X denote functional-secure, functional-insecure, and broken code. Only the final secure policy is returned.

Let x denote the task prompt of a pair. The two sides of a pair often share most of their code, so following LPO (Hasan et al., 2025), we compute the preference loss only on the tokens where they differ. For code y with a difference mask m, which marks the tokens of y that differ from the other side of the pair, we define its masked mean log-likelihood under policy π as

$$\bar{\ell}_\pi(y \mid x, m) = \frac{\sum_t m_t \log \pi(y_t \mid x, y_{<t})}{\max(1, \sum_t m_t)}. \tag{1}$$

For chosen code y^+ and rejected code y^-, the objective combines this preference loss with a cross-entropy loss that trains the policy to generate the differing tokens of the chosen code.

$$\mathcal{L}_{\text{pref}}(\pi) = -\mathbb{E}\left[\log \sigma\left(\beta\left[\bar{\ell}_\pi(y^+ \mid x, m^+) - \bar{\ell}_\pi(y^- \mid x, m^-)\right] - \gamma\right)\right], \tag{2}$$

$$\mathcal{L}_{\text{CE}}(\pi) = -\frac{\sum_{\mathcal{B}} \sum_t m_t^+ \log \pi(y_t^+ \mid x, y_{<t}^+)}{\max\left(1, \sum_{\mathcal{B}} \sum_t m_t^+\right)}, \tag{3}$$

$$\mathcal{L}_{\text{LP}}(\pi) = \mathcal{L}_{\text{pref}}(\pi) + \alpha_{\text{CE}}\, \mathcal{L}_{\text{CE}}(\pi). \tag{4}$$

Here, β scales the likelihood difference, γ sets the preference margin, and α_{CE} weights the cross-entropy loss. The preference loss is a localized form of SimPO (Meng et al., 2024) and is averaged over the pairs in a batch, whereas the cross-entropy loss sums over the differing tokens of all chosen code in a batch $\mathcal{B}$ and divides by their total number. The difference masks mark edited tokens rather than verified vulnerability locations.

For each bootstrap pair, $\pi_{\textbf{sec}}$ uses the secure implementation as y^+ and the insecure implementation as y^-, and $\pi_{\textbf{insec}}$ uses the two swapped. Starting from the same base model gives both policies the same initial coding ability, while separate adapters allow $\pi_{\textbf{insec}}$ to be updated alongside $\pi_{\textbf{sec}}$.

3.2 Pair Construction

Replay tasks come from SeCodePLT (Nie et al., 2025) and CWEval (Peng et al., 2025). Each replay task x consists of a programming prompt, which may include security requirements from the data source, together with executable functionality and security tests. Whenever a pair pool is built, both policies generate rollouts for the task window. The role of a policy determines which outcome it prefers, while the tests determine the label of the code.

For code y, let $F(x, y)$ and $S(x, y)$ indicate that y passes the functionality and security tests, respectively. Scored code receives one of three labels.

$$q(x, y) = \begin{cases} \text{functional-secure}, & F(x, y) \wedge S(x, y), \\ \text{functional-insecure}, & F(x, y) \wedge \neg S(x, y), \\ \text{broken}, & \neg F(x, y). \end{cases} \tag{5}$$

Because both functional labels require passing the functionality tests, generated code cannot count as secure or insecure simply because it fails to run. When the tests cannot produce a verdict, the code is left unscored and is excluded from pair construction.

Each time a task is sampled, we build at most one pair (y_s, y_i) of functional-secure code and functional-insecure code. The functional-secure side is taken first from $\pi_{\textbf{sec}}$ and the functional-insecure side first from $\pi_{\textbf{insec}}$, so that each policy supplies the code it is trained to produce. If the preferred policy generates no qualifying code, we use a rollout of the other policy, so the insecure side can then come from $\pi_{\textbf{sec}}$'s own rollouts. Every pair must contain code generated for the current pair pool. When several rollouts of $\pi_{\textbf{insec}}$ qualify, we choose among them with hard-negative mining, which prefers code with high likelihood under the current $\pi_{\textbf{sec}}$ and small differences from the secure side. Fallback reference code retains its source-provided labels.

3.3 Phase 1: Localized Preference Learning

Phase 1 applies the objective in Equation 4 to the current pair pool P_r, which both policies share, using the same coefficients as the bootstrap. This pool is built from rollouts generated before Phase 2 of the previous round, or before the first round. Unlike the bootstrap pairs, which are fixed and provided by the data sources, these pairs are rebuilt in every round from the rollouts of both policies, and their generated code is labeled by the tests (§3.2). Because the two sides are generated independently, they can differ more than bootstrap pairs, so the difference masks can cover a larger share of each program. Both policies learn from the same pairs, but in opposite directions. $\pi_{\textbf{sec}}$ uses $(y^+, y^-) = (y_s, y_i)$ and learns to prefer the functional-secure code of each pair, while $\pi_{\textbf{insec}}$ uses $(y^+, y^-) = (y_i, y_s)$ and learns to prefer the functional-insecure code. Training $\pi_{\textbf{insec}}$ on the same pairs trains it to prefer the functional-insecure code that $\pi_{\textbf{sec}}$ learns to reject.

3.4 Phase 2: Localized GRPO

After Phase 1, the rollouts generated for the task window W_r build the pair pool P_{r+1} for the next round (§3.2) and are also used to select tasks for each policy in Phase 2. Each policy selects its tasks from its own rollouts. Task filtering is described in Appendix A.

Unlike Phase 1, which uses pairs shared by both policies, each policy samples new rollouts for its kept tasks, separate from those used for pair construction and task selection, and is updated with GRPO (Shao et al., 2024) on its own rollouts. The reward of code y for task x depends only on its label $q(x, y)$ in Equation 5. For $\pi_{\textbf{sec}}$, the reward is

$$r_{\text{sec}}(x, y) = \begin{cases} r_{\text{high}}, & q(x, y) = \text{functional-secure}, \\ r_{\text{mid}}, & q(x, y) = \text{functional-insecure}, \\ r_{\text{low}}, & q(x, y) = \text{broken}, \end{cases} \tag{6}$$

with $r_{\text{high}} > r_{\text{mid}} > r_{\text{low}}$. $\pi_{\textbf{insec}}$ uses the same reward with the two functional labels swapped, so the two policies are rewarded in opposite directions for functional code, while broken code receives the lowest reward for both, so that neither policy can raise its reward by giving up functionality. Because prompts may include security requirements, $\pi_{\textbf{insec}}$ is thereby trained to produce insecure code even when the prompt asks for secure code. As in GRPO, the loss also includes a KL penalty toward the policy at the start of Phase 2. Unscored rollouts are excluded from the group reward statistics and receive zero advantage, so only the KL penalty can affect them.

In localized GRPO, let y^{ref} be the task's reference implementation. We define a reference mask m^{ref} over the tokens of a rollout y,

$$m_t^{\text{ref}} = \mathbb{1}\left[\, y_t \text{ is not aligned with } y^{\text{ref}} \text{ by a token-level diff} \,\right], \tag{7}$$

and compute the GRPO loss of y only on tokens with $m_t^{\text{ref}} = 1$. Unlike the pair mask in Equation 1, this mask compares a rollout with a reference implementation rather than with the other side of a pair. This localizes program-level rewards without requiring token-level security labels; like the pair mask, the reference mask marks differing tokens rather than verified vulnerability locations. The updated adapters start the next round.

4 EVALUATION

4.1 RESEARCH QUESTIONS

We organize the evaluation around four questions. These questions distinguish measured benchmark outcomes from the additional evidence needed to explain or generalize them.

RQ1: Overall Effectiveness. Does our method improve secure code generation while preserving functionality?

RQ2: Component Contributions. How does each training component affect performance?

RQ3: Negative-Source Utility. Does the evolving insecure policy sustain useful hard negatives?

RQ4: OOD Generalization. Does replay improve performance on languages and CWE categories absent from replay data, relative to Bootstrap $\pi_{\mathbf{sec}}$?

4.2 EXPERIMENTAL SETUP

Datasets and Benchmarks. For training, we use secure and insecure code pairs from SafeCoder and SeCodePLT to bootstrap both policies. During replay, we use programming tasks from SeCodePLT and CWEval. Their executable tests check the generated code and provide labels and rewards for learning.

For evaluation, we test the secure policy on SecCodeBench and EvalPlus. SecCodeBench tests functionality and security on 98 tasks, and its correct-and-secure rate (C&S) counts programs that pass both tests. Each SecCodeBench task targets one CWE. We split the benchmark by CWE into a validation set of 29 tasks from six CWEs, used only to choose training rounds, and a test set of the other 69 tasks from 15 CWEs, on which we report results; no CWE appears in both. EvalPlus tests functional correctness on 542 tasks to measure general coding ability. Neither benchmark supplies training examples or rewards. Appendix A.4 details the split and the round selection.

Baselines. We compare against the base model and three types of security methods. SafeCoder (He et al., 2024) and DiSCo-LPO (Hasan et al., 2025) learn from fixed examples; DiSCo-LPO also focuses preference learning on code differences. TSP (Chen et al., 2026) instead learns from alternatives generated by a previous version of the learner, providing a self-play comparison without a separately trained insecure policy. Finally, RESCUE (Shi & Zhang, 2026) retrieves security guidelines and code examples without updating model weights, testing an alternative to training.

Ablation Study. We test the role of the insecure policy and the two training phases with four ablations. No $\pi_{\mathbf{insec}}$ removes the insecure policy, while Frozen $\pi_{\mathbf{insec}}$ fixes it after bootstrap but continues sampling, testing whether its updates help. Preference-only removes online updates to test their contribution. GRPO-only uses online updates without replay preference learning or the insecure policy; it therefore tests both changes together rather than preference learning alone. We compare all variants with the full method on security and functionality.

Training and Evaluation Settings. We tune hyperparameters for our method and the training-based baselines: SafeCoder, DiSCo-LPO, and TSP. We report mean scores over three training seeds. Base and RESCUE require no training and use one evaluation per model. For our method and each ablation, we select one replay round shared by all seeds by its mean validation C&S, with ties broken by mean EvalPlus accuracy and then the earlier round. Appendix A details the tuning and evaluation procedures.

Table 1: Security and functionality across five models (%). SecCodeBench columns cover the 69 test-split tasks; EvalPlus covers all 542 tasks. Rounds of the full method and the ablations are selected on the 29 validation-split tasks. Trained methods average three seeds; Base and RESCUE use one evaluation. Parentheses show percentage-point changes from Base (▲: increase; ▼: decrease). Within each model and metric, bold marks the best non-base score and darker blue indicates higher scores. Dashes denote unavailable results.

	SecCodeBench			EvalPlus
Method	Func.↑	Sec.↑	C&S↑	Func.↑
Granite 4.1 (8B)				
Base	55.1	20.3	15.9	74.5
SafeCoder	47.3 (▼ 7.7)	28.0 (▲ 7.7)	18.8 (▲ 2.9)	67.9 (▼ 6.6)
DiSCo-LPO	48.3 (▼ 6.8)	28.0 (▲ 7.7)	16.4 (▲ 0.5)	73.2 (▼ 1.4)
TSP	52.7 (▼ 2.4)	23.7 (▲ 3.4)	14.0 (▼ 1.9)	**75.8** (▲ 1.3)
RESCUE	**53.6** (▼ 1.4)	26.1 (▲ 5.8)	15.9 (0.0)	74.7 (▲ 0.2)
Full method (selected)	44.4 (▼ 10.6)	**35.7** (▲ 15.5)	**24.6** (▲ 8.7)	73.8 (▼ 0.7)
No π_{insec}	44.9 (▼ 10.1)	29.0 (▲ 8.7)	18.8 (▲ 2.9)	72.6 (▼ 2.0)
Frozen π_{insec}	42.5 (▼ 12.6)	30.0 (▲ 9.7)	20.8 (▲ 4.8)	72.9 (▼ 1.6)
Preference-only	31.9 (▼ 23.2)	27.5 (▲ 7.2)	17.9 (▲ 1.9)	70.3 (▼ 4.2)
GRPO-only (no π_{insec})	44.0 (▼ 11.1)	29.0 (▲ 8.7)	20.3 (▲ 4.3)	73.7 (▼ 0.8)
Qwen2.5-Coder (7B)				
Base	53.6	29.0	18.8	74.4
SafeCoder	53.6 (0.0)	36.7 (▲ 7.7)	22.2 (▲ 3.4)	68.6 (▼ 5.8)
DiSCo-LPO	46.9 (▼ 6.8)	36.2 (▲ 7.2)	26.6 (▲ 7.7)	71.0 (▼ 3.4)
TSP	**56.0** (▲ 2.4)	31.4 (▲ 2.4)	20.8 (▲ 1.9)	74.2 (▼ 0.1)
RESCUE	55.1 (▲ 1.4)	39.1 (▲ 10.1)	21.7 (▲ 2.9)	**75.1** (▲ 0.7)
Full method (selected)	37.7 (▼ 15.9)	**43.0** (▲ 14.0)	27.5 (▲ 8.7)	70.8 (▼ 3.5)
No π_{insec}	53.6 (0.0)	40.6 (▲ 11.6)	**28.5** (▲ 9.7)	73.2 (▼ 1.2)
Frozen π_{insec}	36.2 (▼ 17.4)	41.1 (▲ 12.1)	24.2 (▲ 5.3)	72.9 (▼ 1.5)
Preference-only	34.3 (▼ 19.3)	39.6 (▲ 10.6)	24.6 (▲ 5.8)	70.4 (▼ 4.0)
GRPO-only (no π_{insec})	54.6 (▲ 1.0)	35.7 (▲ 6.8)	26.1 (▲ 7.2)	71.0 (▼ 3.3)
Qwen2.5-Coder (14B)				
Base	60.9	31.9	23.2	78.6
SafeCoder	58.0 (▼ 2.9)	39.6 (▲ 7.7)	28.0 (▲ 4.8)	74.4 (▼ 4.2)
DiSCo-LPO	46.4 (▼ 14.5)	45.9 (▲ 14.0)	28.5 (▲ 5.3)	71.6 (▼ 7.0)
TSP	59.4 (▼ 1.4)	29.5 (▼ 2.4)	20.8 (▼ 2.4)	78.6 (0.0)
RESCUE	63.8 (▲ 2.9)	42.0 (▲ 10.1)	30.4 (▲ 7.2)	**78.8** (▲ 0.2)
Full method (selected)	60.4 (▼ 0.5)	47.3 (▲ 15.5)	33.8 (▲ 10.6)	76.9 (▼ 1.7)
No π_{insec}	**64.3** (▲ 3.4)	40.1 (▲ 8.2)	30.0 (▲ 6.8)	76.2 (▼ 2.4)
Frozen π_{insec}	61.4 (▲ 0.5)	**49.8** (▲ 17.9)	**38.2** (▲ 15.0)	76.8 (▼ 1.8)
Preference-only	58.9 (▼ 1.9)	46.9 (▲ 15.0)	34.8 (▲ 11.6)	76.4 (▼ 2.2)
GRPO-only (no π_{insec})	54.1 (▼ 6.8)	40.1 (▲ 8.2)	29.5 (▲ 6.3)	76.7 (▼ 1.9)
Phi-4 Mini (3.8B)				
Base	39.1	23.2	14.5	59.8
SafeCoder	27.1 (▼ 12.1)	26.1 (▲ 2.9)	15.5 (▲ 1.0)	58.7 (▼ 1.0)
DiSCo-LPO	17.9 (▼ 21.3)	24.2 (▲ 1.0)	11.6 (▼ 2.9)	50.3 (▼ 9.5)
TSP	**41.5** (▲ 2.4)	25.6 (▲ 2.4)	**16.4** (▲ 1.9)	**60.8** (▲ 1.0)
RESCUE	39.1 (0.0)	18.8 (▼ 4.3)	13.0 (▼ 1.4)	59.4 (▼ 0.4)
Full method (selected)	30.9 (▼ 8.2)	29.0 (▲ 5.8)	14.5 (0.0)	58.7 (▼ 1.1)
No π_{insec}	30.9 (▼ 8.2)	29.0 (▲ 5.8)	14.5 (0.0)	58.3 (▼ 1.5)
Frozen π_{insec}	33.3 (▼ 5.8)	**30.9** (▲ 7.7)	15.0 (▲ 0.5)	57.6 (▼ 2.2)
Preference-only	34.3 (▼ 4.8)	28.0 (▲ 4.8)	15.9 (▲ 1.4)	59.4 (▼ 0.4)
GRPO-only (no π_{insec})	31.9 (▼ 7.2)	29.5 (▲ 6.3)	13.5 (▼ 1.0)	57.6 (▼ 2.2)
CodeLlama (7B)				
Base	34.8	24.6	14.5	40.8
SafeCoder	35.3 (▲ 0.5)	23.7 (▼ 1.0)	15.5 (▲ 1.0)	41.9 (▲ 1.2)
DiSCo-LPO	23.7 (▼ 11.1)	21.3 (▼ 3.4)	12.6 (▼ 1.9)	36.3 (▼ 4.5)
TSP	35.3 (▲ 0.5)	21.7 (▼ 2.9)	15.0 (▲ 0.5)	40.5 (▼ 0.2)
RESCUE	37.7 (▲ 2.9)	24.6 (0.0)	14.5 (0.0)	42.3 (▲ 1.5)
Full method (selected)	**39.6** (▲ 4.8)	**28.5** (▲ 3.9)	**15.9** (▲ 1.4)	44.2 (▲ 3.4)
No π_{insec}	30.4 (▼ 4.3)	26.6 (▲ 1.9)	13.5 (▼ 1.0)	**46.3** (▲ 5.5)
Frozen π_{insec}	32.9 (▼ 1.9)	22.7 (▼ 1.9)	11.1 (▼ 3.4)	43.8 (▲ 3.1)
Preference-only	31.9 (▼ 2.9)	23.7 (▼ 1.0)	12.6 (▼ 1.9)	42.9 (▲ 2.1)
GRPO-only (no π_{insec})	32.4 (▼ 2.4)	24.2 (▼ 0.5)	10.6 (▼ 3.9)	44.3 (▲ 3.5)

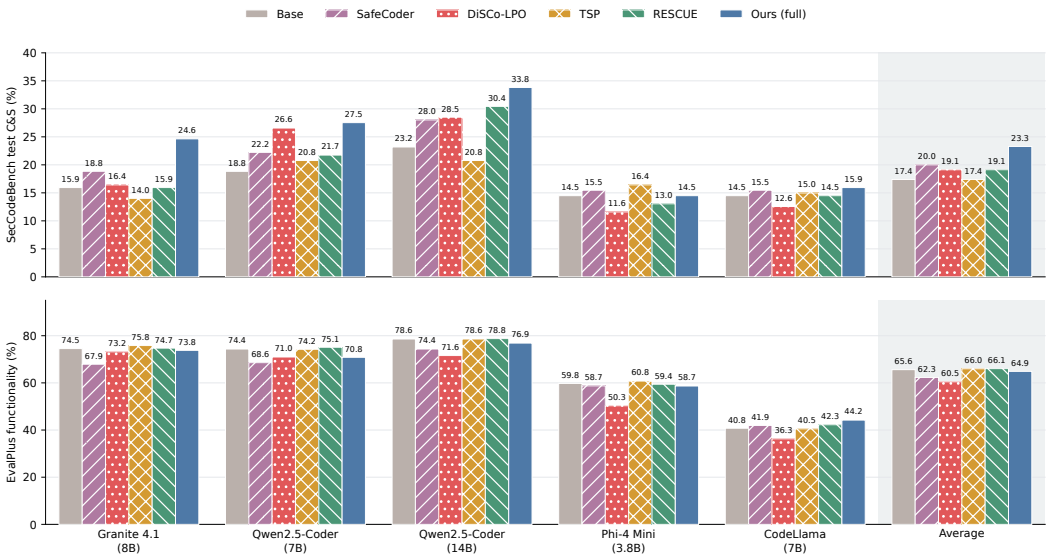

Figure 3: C&S and EvalPlus comparison for Base, SafeCoder, DiSCo-LPO, TSP, RESCUE, and the full method (Ours), using the same selected checkpoints and test-split scores as Table 1. Trained methods average their three seeds; Base and RESCUE each have one evaluation. The rightmost group averages the five models with equal weight. Labels give percentages, and bar axes start at zero.

4.3 RQ1: Overall Effectiveness

EvoSecCoder achieves the highest mean C&S among the compared baselines (Table 1, Figure 3): 23.3%, compared with 17.4% for the base models and 17.4–20.0% for the baselines. It outperforms DiSCo-LPO and RESCUE on all five models, and SafeCoder and TSP on four; on Phi-4 Mini, both lead and EvoSecCoder matches the base model. Mean EvalPlus accuracy is 64.9%, above SafeCoder and DiSCo-LPO but below the base models' 65.6%. CodeLlama improves in functionality, while the other four models decline.

4.4 RQ2: Component Contributions

The full method achieves the highest mean C&S across the five models (Table 1): 23.3%, compared with 21.1% for No π_{insec}, 21.8% for Frozen π_{insec}, 21.2% for preference-only, and 20.0% for GRPO-only. It outperforms GRPO-only on all five models. However, No π_{insec} scores higher on Qwen 7B and ties on Phi-4 Mini; Frozen π_{insec} and preference-only score higher on Qwen 14B and Phi-4 Mini.

GRPO-only also removes π_{insec}, so its comparison with the full method does not isolate the contribution of preference learning. Mean EvalPlus accuracy ranges from 63.9% to 65.3% across these variants, with the full method at 64.9%.

The mean C&S advantage varies with the CWE split. Across all 22,922 CWE-disjoint 29/69 splits, the full method exceeds each baseline, No π_{insec}, and GRPO-only in at least 97.8% of splits, Frozen π_{insec} in 93.7%, and preference-only in 75.9% (Appendix A.4). Figure 5 shows test-split trajectories for the full method, No π_{insec}, and Frozen π_{insec}.

4.5 RQ3: Negative-Source Utility

We evaluate π_{insec} on 300 fixed replay tasks, stratified by source, language, and first CWE (261 SeCodePLT and 39 CWEval tasks). Each checkpoint generates one response per task, scored by the replay's executable tests. Valid negatives pass functionality tests but fail security tests.

At R5 and R10, all five models exceed their bootstrap functional-and-insecure rates (Figure 4). Across models, these rates rise from 18.9–30.2% after bootstrap to 32.2–55.0% at R10, compared with 9.0–13.3% for the untuned base. Functional rates also increase from bootstrap to R10 for all five

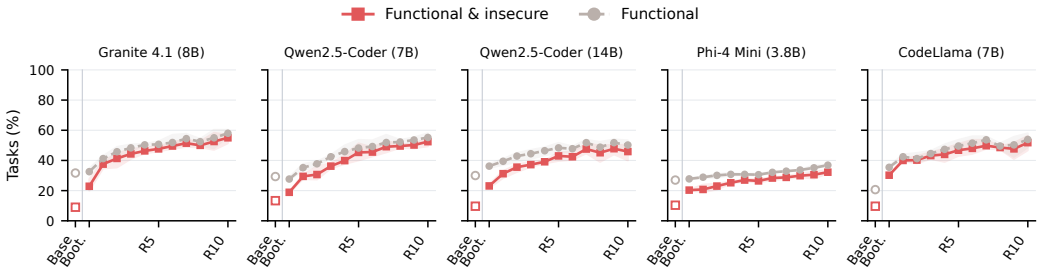

Figure 4: Insecure-policy outcomes on a fixed 300-task replay sample. Red: functional-and-insecure rate; gray: functional rate. $\pi_{\mathbf{insec}}$ is evaluated after bootstrap and every replay round of the full method; hollow markers show the untuned base. Lines average three seeds; bands span their observed range. All checkpoints use the same tasks, which were seen in training.

models. These results show increased valid-negative supply on training tasks, but do not establish difficulty for $\pi_{\mathbf{sec}}$, diversity, or generalization; memorization may contribute to the gains. Appendix B reports training diagnostics and a separate held-out evaluation of $\pi_{\mathbf{insec}}$.

4.6 RQ4: OOD Generalization

We compare Bootstrap $\pi_{\mathbf{sec}}$ with the selected $\pi_{\mathbf{sec}}$ checkpoint on categories absent from the supplied replay corpus, matching seeds and evaluation tasks. Table 2 reports the results on the test split: 32 tasks in a replay-unseen language and 19 tasks with replay-unseen target CWEs. Appendix C lists the unseen CWE categories and details the subsets and aggregation. Java C&S increases for three models, is unchanged for Phi-4 Mini, and declines slightly for Qwen 14B; replay-unseen CWE C&S increases for four models and declines for Qwen 14B. Qwen 7B loses functionality on both subsets. This comparison measures changes over replay; categories seen during bootstrap remain eligible.

Table 2: Replay transfer on test-split tasks: C&S (%), averaged over three seeds. Δ: Ours minus Bootstrap (pp).

	Unseen Lang ($N = 32$)			Unseen CWE ($N = 19$)		
Model	Boot.	Ours	**Δ** (pp)	Boot.	Ours	**Δ** (pp)
Granite 4.1 (8B)	18.75	26.04	**+7.29**	31.58	33.33	**+1.75**
Qwen2.5-Coder (7B)	32.29	38.54	**+6.25**	31.58	35.09	**+3.51**
Qwen2.5-Coder (14B)	38.54	37.50	**-1.04**	50.88	45.61	**-5.26**
Phi-4 Mini (3.8B)	14.58	14.58	**0.00**	28.07	29.82	**+1.75**
CodeLlama (7B)	15.62	19.79	**+4.17**	21.05	29.82	**+8.77**

5 Conclusion

We introduced EvoSecCoder, which trains two adapters over a shared base toward opposite security objectives using localized preference learning and online reinforcement learning with rewards from executable tests.

Across five models, rounds selected on validation CWEs achieve the highest mean test C&S among the compared methods, with functionality losses and exceptions for individual models. Longer replay can reduce C&S, and round selection depends on the CWE split. Improvements over bootstrap on replay-unseen categories also vary by model. Configurations were tuned on the full benchmarks; selection-independent transfer and label efficiency remain to be established. Future work should measure the diversity of negative examples and label efficiency under matched compute budgets and evaluate transfer to languages and CWEs excluded from training.

AI USE STATEMENT

In this work, we used generative AI tools to aid and polish writing, for retrieval and discovery, for research ideation and execution, to draft parts of the paper, and to assist with translation. We have not used generative AI tools for generating synthetic datasets, and proving mathematical claims is not applicable to this work. Additionally, we used generative AI tools to revise the manuscript, to search for related work, to automate the execution of experiment scripts, and to create initial drafts of figures. We have reviewed all AI-assisted work. Two authors verified every citation against its original source, the authors read and checked all translated text before submission, all results from automated runs were checked by the authors against the raw experiment logs, and the authors revised all AI-drafted figures to ensure they accurately depict the method and reported results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

EVOSECCODER aims to improve secure code generation by learning from secure and insecure implementations. It deliberately trains $\pi_{\mathbf{insec}}$ to generate functional but insecure code as supervision for security learning. Although only $\pi_{\mathbf{sec}}$ is needed for deployment, $\pi_{\mathbf{insec}}$ and its generated code could be misused to introduce vulnerabilities into software. This risk should inform decisions about whether and how to release insecure checkpoints and generated examples.

Our security labels reflect outcomes under the specified executable tests, not guarantees that programs are free of vulnerabilities. Benchmark improvements may also accompany losses in functionality and do not establish reliability in deployment settings. Generated code therefore requires independent functional testing and security review before use. The intended role of EVOSECCODER is to support secure software development, while retaining human responsibility for reviewing and deploying code.

REFERENCES

Wenqi Chen, Ziyan Zhang, Bin Wang, Lin Liu, Hengheng Zhang, and Zhengsu Chen. Learn from your mistakes: Tree-like self-play for secure code LLMs. arXiv:2606.03489, version 2, 2026. URL https://arxiv.org/abs/2606.03489v2.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 6621–6642, 2024. URL https://proceedings.mlr.press/v235/chen24j.html.

Hossein Hajipour, Lea Schönherr, Thorsten Holz, and Mario Fritz. HexaCoder: Secure code generation via oracle-guided synthetic training data. arXiv:2409.06446, 2024. URL https://arxiv.org/abs/2409.06446.

Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. Teaching an old LLM secure coding: Localized preference optimization on distilled preferences. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26039–26057, 2025. doi: 10.18653/v1/2025.acl-long.1263. URL https://aclanthology.org/2025.acl-long.1263/.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023. doi: 10.1145/3576915.3623175. URL https://dl.acm.org/doi/10.1145/3576915.3623175.

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

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International*

Conference on Learning Representations, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract.html.

Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, Yuxiang Wei, Lingming Zhang, Ismini Lourentzou, and Gang Wang. PurpCode: Reasoning for safer code generation. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/4f697305ef1f868ad77c3c0027989a6f-Abstract-Conference.html.

Yu Meng, Mengzhou Xia, and Danqi Chen. SimPO: Simple preference optimization with a reference-free reward. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://papers.neurips.cc/paper_files/paper/2024/hash/e099c1c9699814af0be873a175361713-Abstract-Conference.html.

Yuzhou Nie, Zhun Wang, Yu Yang, Ruizhe Jiang, Yuheng Tang, Xander Davies, Yarin Gal, Bo Li, Wenbo Guo, and Dawn Song. SeCodePLT: A unified benchmark for evaluating the security risks and capabilities of code GenAI. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/13d0a982aae786d473f6949b734e2720-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

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. doi: 10.1109/SP46214.2022.9833571. URL https://ieeexplore.ieee.org/document/9833571.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. CWEval: Outcome-driven evaluation on functionality and security of LLM code generation. In *IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, 2025. URL https://ieeexplore.ieee.org/document/11028476.

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:2602.23407, 2026. URL https://arxiv.org/abs/2602.23407.

SecCodeBench Team. SecCodeBench: A benchmark suite for evaluating security of AI-generated code. Benchmark repository, 2025. URL https://github.com/alibaba/sec-code-bench. Accessed September 9, 2026.

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:2402.03300, 2024. URL https://arxiv.org/abs/2402.03300.

Jiahao Shi and Tianyi Zhang. RESCUE: Retrieval augmented secure code generation. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/7eb9349690db0925a9fb09dbe2ee7cf0-Abstract-Conference.html.

Tianyi Wu, Mingzhe Du, Yue Liu, Chengran Yang, Terry Yue Zhuo, Jiaheng Zhang, and See-Kiong Ng. Secure code generation via online reinforcement learning with vulnerability reward model. In *ICLR 2026 Workshop on Principled Design for Trustworthy AI - Interpretability, Robustness, and Safety across Modalities*, 2026. URL https://openreview.net/forum?id=zifVKgTAK4.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. ProSec: Fortifying code LLMs with proactive security alignment. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69689–69704, 2025. URL https://proceedings.mlr.press/v267/xu25aa.html.

Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui, Li Zhong, Xiaohan Fu, Haohui Mai, Vish Krishnan, Jianfeng Gao, and Jingbo Shang. Training language models to generate quality code with program analysis feedback. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c676042ee324ba0bc60b8fcf2278a0cd-Paper-Conference.pdf.

A Training and Evaluation Details

A.1 Training Procedure

Preference-update coverage. We compare the number of pairs in the pool with the number of replay tasks sampled to build it. If the pool is empty or this ratio falls below a threshold, we skip Phase 1 in that round, so that a small number of pairs does not drive an entire update. Pair construction and Phase 2 still proceed in that round.

Online task selection. The reward-variance task filter keeps a replay task for a policy when that policy's scored rollouts receive more than one distinct reward, so that the policy's outcomes on the task vary. If too few tasks remain, the policy uses the whole task window instead. The kept tasks can include tasks that yield no pair.

Reference selection. Each policy uses the reference implementation that matches its role and falls back to the other one when it is unavailable. If no reference implementation is available, all generated tokens of the rollout are used.

A.2 Hyperparameter Tuning

Scope and comparison rule. We tune our method, SafeCoder, and DiSCo-LPO separately for each base model. For these methods, we rank candidate configurations by mean SecCodeBench C&S, then by mean EvalPlus functional correctness. For replay, we compare the same configuration at the same round across seeds, rather than choosing a different best round for each seed. Only candidates with comparable scoring and all required seed results enter the comparison. Because both benchmarks guided this tuning, which predates the validation/test split of Appendix A.4 and used all 98 SecCodeBench tasks, the reported test-split scores are not fully untouched at the configuration level.

Search and reporting seeds. SafeCoder and DiSCo-LPO use seeds 42 and 43 during tuning. Our method also uses these two seeds for Granite, Qwen 7B, Qwen 14B, and Phi-4 Mini. After choosing a configuration, we run it with seed 44 and report the three-seed mean; we do not repeat the whole search with that seed. For CodeLlama, our method instead uses seeds 42, 43, and 44 throughout its final search. Thus, three-seed reporting does not imply that every candidate was tested with three seeds.

Our method. We vary the preference margin and the learning rates for preference learning and online learning. The searches use model-specific candidate combinations, rather than one common full grid. The Granite search refines a shortlist and extends replay from five to ten rounds. The Qwen 7B search combines a small grid with additional candidate settings. For Qwen 14B, Phi-4 Mini, and CodeLlama, the broad search specifies 16 combinations that cover pairs of values across the three parameters. We consider available checkpoints throughout replay, including intermediate rounds. Incomplete runs do not supply missing seed results, and we do not assume that the final round is best.

SafeCoder. We search a 4×4 grid over the learning rate and KL loss weight. For Granite, both Qwen models, and Phi-4 Mini, the learning rates are $\{5 \times 10^{-6}, 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}\}$. CodeLlama instead uses $\{2.5 \times 10^{-4}, 5 \times 10^{-4}, 10^{-3}, 2 \times 10^{-3}\}$. All five models use KL weights $\{0.4, 0.8, 1.6, 3.2\}$. We rank the completed configurations using the two-seed mean scores described above.

DiSCo-LPO. We search a 4×4 grid over the learning rates for supervised fine-tuning and localized preference learning. The supervised rates are $\{2.5 \times 10^{-5}, 5 \times 10^{-5}, 10^{-4}, 2 \times 10^{-4}\}$, and the preference-learning rates are $\{2.5 \times 10^{-6}, 5 \times 10^{-6}, 10^{-5}, 2 \times 10^{-5}\}$. We use the same grid for all five models and rank completed configurations by the two-seed means.

TSP and methods without training. For TSP, we test learning rates $\{10^{-5}, 5 \times 10^{-6}, 10^{-6}, 10^{-7}\}$. Most runs at the higher rates suffer training collapse; only 10^{-7} yields stable results, so we use it for the reported comparison. We keep three epochs per iteration, two iterations, and a DPO coefficient of 0.1. This search chooses the learning rate based on training stability. Base and RESCUE do not update model weights, so neither requires training hyperparameter tuning.

A.3 Experimental Settings and Checkpoint Selection

Trained methods use seeds 42, 43, and 44; Base and RESCUE each use one evaluation. The TSP baseline uses a learning rate of 10^{-7}. Our method, SafeCoder, and DiSCo-LPO use the configurations chosen by the tuning procedure above. TSP uses the learning rate chosen through the stability-based search above. For the full method and each component ablation, we choose the round available for all three seeds with the highest mean C&S on the SecCodeBench validation split (Appendix A.4), breaking ties by mean EvalPlus functionality and then the earlier round. Thus, scores average a shared round across seeds, rather than independently selected seed peaks. The selected full-method checkpoints are R2 for Granite, R9 for Qwen 7B, R1 for Qwen 14B, R3 for Phi-4 Mini, and R7 for CodeLlama. Round labels identify checkpoints and do not represent matched compute budgets.

Within each model, all methods use the same response parser. Rates divide successful tasks by the number of tasks in the reported split (69 SecCodeBench test tasks, 542 EvalPlus tasks); all evaluations in the main comparison have complete scoring coverage. Round trajectories average the three seeds at each available round. A point is omitted if any seed is missing, and missing points are not interpolated.

A.4 Round Selection on a CWE-Disjoint Validation Split

Split. Every SecCodeBench task targets exactly one CWE, so we split the benchmark by CWE rather than by task. Starting from the 21 CWEs in numerical order, a pseudorandom shuffle with seed 0 is followed by a greedy pass that adds each CWE to the validation set while the total stays at most 29 tasks ($\mathrm{round}(0.3 \times 98)$); the first shuffle reaching exactly 29 tasks is used. The validation set contains CWE-352, 415, 611, 798, 917, and 918 (29 tasks, 21 in Java); the test set contains the other 15 CWEs (69 tasks). EvalPlus has no CWE labels and is not split. The split is identical for every model and method, and its seed was not tuned. This protocol replaced an earlier selection on all 98 tasks after the main experiments were complete.

Selection and scoring. For the full method and each ablation, we select the round with the highest three-seed mean validation C&S, then the highest mean EvalPlus accuracy, then the earlier round. Only rounds with per-task verdicts for all three seeds are eligible. These verdicts come from a later re-scoring pass over the saved generations; for 199 of the 450 trajectory checkpoints, its benchmark totals differ from the original scores. Table 1 uses these verdicts for every method. Baselines without replay rounds keep their tuned checkpoints and are scored on the same test split.

Selected rounds. Figure 5 shows the test-split trajectories. The full method's validation-selected rounds are R2 for Granite, R9 for Qwen 7B, R1 for Qwen 14B, R3 for Phi-4 Mini, and R7 for CodeLlama. The earlier protocol, which selected rounds on all 98 tasks, chose the same round for CodeLlama and R10, R10, R10, and R2 for the other four; on the test split, those rounds score 1.4 points lower for Granite and 1.4, 0.5, and 1.0 points higher for Qwen 7B, Qwen 14B, and Phi-4 Mini, respectively; the higher scores reflect selection on tasks that are now held out.

Sensitivity to the split. To measure how much these results depend on the particular validation CWEs, we repeat the selection for every CWE-disjoint split with exactly 29 validation tasks (22,922 splits), re-selecting the full method and all four ablations on each. The selected full-method round matches the earlier full-benchmark round in 8%, 37%, 10%, 12%, and 34% of splits for Granite, Qwen 7B, Qwen 14B, Phi-4 Mini, and CodeLlama, so round choice is sensitive to the validation CWEs. The full method's five-model mean test C&S nonetheless exceeds that of Base in 99.8% of splits, SafeCoder in 97.8%, DiSCo-LPO in 98.4%, TSP in 99.6%, and RESCUE in every split, by 2.2 to 3.3 points on average. Against the ablations, it is higher in 98.0% (No $\pi_{\mathbf{insec}}$), 93.7% (Frozen $\pi_{\mathbf{insec}}$), 98.5% (GRPO-only), and 75.9% (preference-only) of splits, with mean margins of 1.7, 1.4, 1.7, and 0.7 points. The advantage over external baselines is therefore robust to the split, whereas the advantage over preference-only is not.

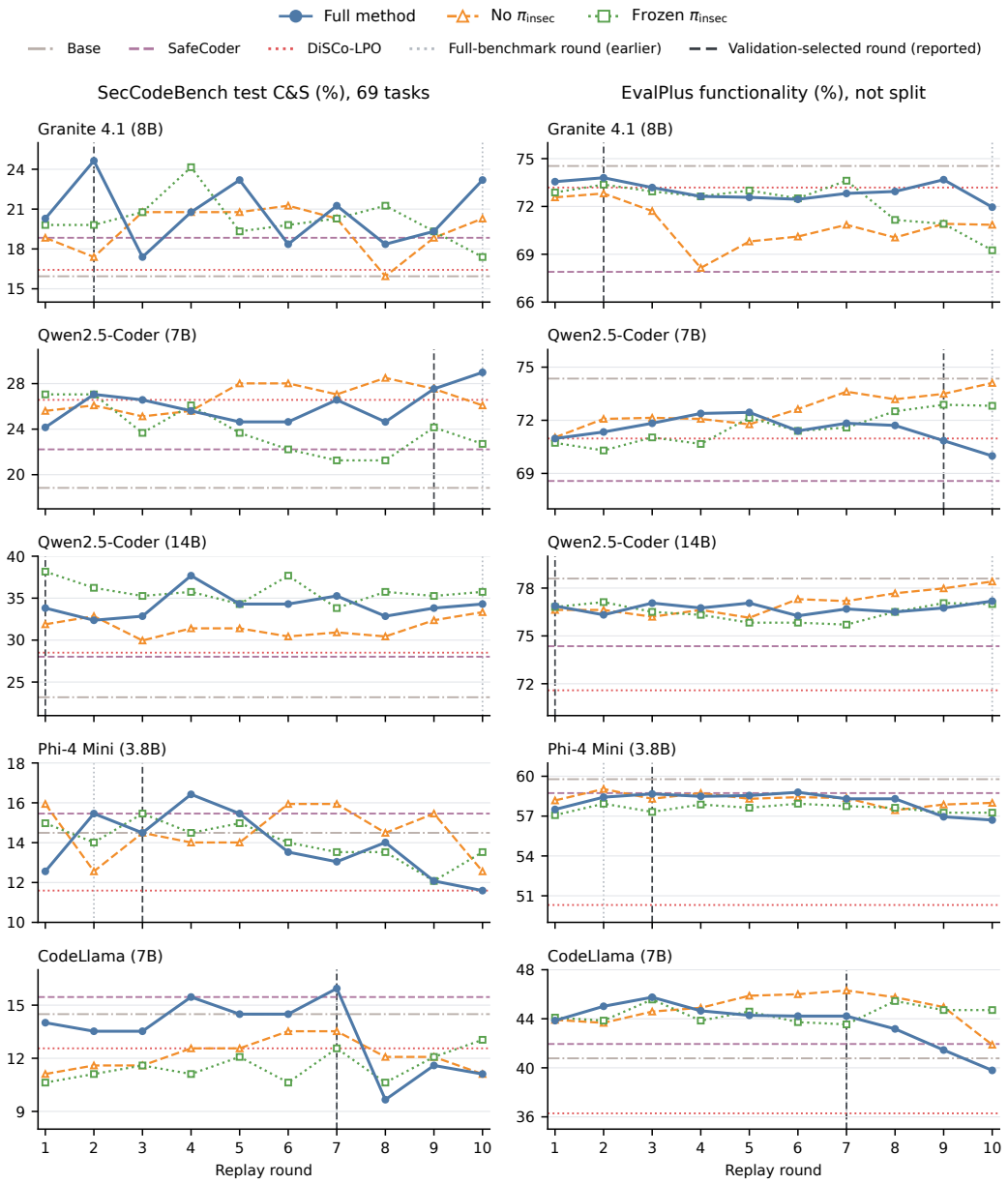

Figure 5: Round trajectories on the 69-task CWE-disjoint SecCodeBench test split (left) and on all of EvalPlus (right), for the full method, No $\pi_{\mathbf{insec}}$, and Frozen $\pi_{\mathbf{insec}}$. Curves are three-seed means of re-scored per-task verdicts; horizontal lines show Base, SafeCoder, and DiSCo-LPO on the same tasks. Dashed vertical lines mark the validation-selected full-method round used in Table 1; dotted lines mark the round chosen earlier on all 98 tasks. Missing paired-seed checkpoints are not interpolated. No test task shares a target CWE with the validation split used to choose the dashed round.

B Replay Data Supply and Negative-Source Diagnostics

This appendix supplements RQ3 with aggregate training records from Full, Frozen π_{insec}, and No π_{insec}. The snapshot contains 450 round records: five base models, three methods, seeds 42/43/44, and R1–R10. We resolve exactly the registered trajectories used in the main evaluation, including resume ancestors, without selecting new checkpoints. Each exported observation records the source `round.json` path and SHA-256 hash; its evaluation scores retain the original catalog provenance, not the separate slice re-evaluation. All plotted means weight seeds equally and require all three seeds. Shaded bands, where shown, span the observed seed minimum and maximum; they are not confidence intervals.

Pool timing and scope. Round labels identify the Phase-1 update that consumes a pair pool. The first update consumes the initial generation pool; subsequent updates consume the carried pool from the preceding round's post-Phase-1 generation pass. These are not evaluations of the final, post-GRPO checkpoint at each round. The round-end continuation state's `pending` pool belongs to the next update and is not substituted for the consumed pool. Task windows change between passes; these curves describe the training distribution rather than a fixed diagnostic test set. Stored round labels also retain the historical numbering discussed in Section 4.

Fresh-pair coverage and provenance. For each seed and round, coverage is the number of assembled pairs containing at least one current-pool generation divided by attempted tasks. Figure 6 partitions attempted tasks into pairs with two current generations, a current generation plus a queued generation, a current generation plus a reference, and tasks yielding no pair. Thus the reference band uses attempted tasks as its denominator, not only successful pairs. These are joint π_{sec} and π_{insec} supply diagnostics; current-pair coverage does not identify π_{insec}'s individual contribution or establish negative difficulty.

Functional generation counts. Figure 7 reports the Full method's stored counts of functional-and-secure and functional-and-insecure generations. The scorer counts these outcomes across both π_{sec} and π_{insec} before pair selection, including only scored generations. The historical aggregate records do not preserve their allocation to π_{sec} versus π_{insec}, or separate totals for nonfunctional and unscored generations. We therefore show pooled counts rather than inventing model-specific curves or four-category percentages. Counts are not success rates: total generation counts are not audited here. The selected replay queue is a filtered subset and cannot recover the missing full generation distribution. Neither pooled curve by itself demonstrates that the two policies specialize in opposite directions.

Per-policy GRPO rewards. Phase-2 GRPO training logs separate the two policies. Each round trains π_{sec} and then π_{insec} with its own verifiable reward: 1 for functional code of the policy's target polarity (secure for π_{sec}, insecure for π_{insec}), 0.3 for functional code of the opposite polarity, and 0 for nonfunctional code. The offline W&B records attribute each logged step to its trainer through the recorded output directory, and rounds copied from a resume parent are accepted only when the adapter file is identical. For each seed and round we average the five logged optimizer steps of the 50-step update and record the logged fraction of rollout groups whose rewards have zero variance. π_{insec}'s mean reward exceeds π_{sec}'s in 8–10 of the ten rounds for every base model. From R1 to R10 it rises from 0.58 to 0.74 for Granite, 0.45 to 0.65 for Qwen 7B, and 0.59 to 0.65 for Qwen 14B, with π_{insec}'s zero-variance fraction rising alongside; it stays near 0.5 for Phi-4 Mini and varies between 0.40 and 0.69 without a clear trend for CodeLlama. These rewards are measured on π_{insec}'s own rollouts over the prompts that survive dynamic sampling, which removes groups with uniform rewards before each update and changes across rounds. They therefore mix the 1.0 and 0.3 outcomes, are not functional-and-insecure rates, and cannot be compared with held-out scores or across rounds as a fixed-task measurement.

Insecure-policy transfer. We evaluate π_{insec} itself on a fixed task set. For each locked Full seed we generate once from π_{insec} after bootstrap and after every replay round, and from the untuned base model, on a fixed 300-task sample of the 1,004-task replay pool shared by all runs, stratified by source, language, and first CWE (261 SeCodePLT and 39 CWEval tasks) and scored by the replay's executable oracles. π_{insec} after bootstrap, R5, and R10 is also evaluated on the held-out SecCodeBench/EvalPlus suite with each model's standard decoding and response parser. A task counts as functional and insecure only when it passes the functional oracle and fails the security

oracle, so nonfunctional code is never counted as insecure. On the replay sample, $\pi_{\mathbf{insec}}$'s functional-and-insecure rate rises across rounds for every model, with only small round-to-round dips, from 9.0–13.3% for the base to 32.2–55.0% at R10, and its functional rate rises with it (for example, from 31.7% to 58.0% for Granite). On held-out SecCodeBench, the same checkpoints end below the base for every model; for example, the rate falls from 36.7% to 26.9% for Granite and from 38.8% to 28.2% for Qwen 7B. Much of this held-out decline consists of Java code that fails to build. From bootstrap to R10, mean build failures per seed rise from 9.3 to 25.7 for Granite, from 11.3 to 27.7 for Qwen 7B, and from 18.7 to 34.7 for CodeLlama. In inspected Qwen 7B outputs, the standard parser dropped the package declaration from unfenced Java responses; Granite's $\pi_{\mathbf{insec}}$ omitted the declaration itself. $\pi_{\mathbf{insec}}$ thus learns its training objective on the replay distribution, but this behavior does not transfer to the held-out benchmark. The replay sample consists of tasks seen during training, so its gain may partly reflect memorization. When $\pi_{\mathbf{insec}}$'s held-out suite was scored, the proxy serving the original SecCodeBench judge was unavailable, so its LLM-judged Java tasks used `claude-sonnet-4-6`, whereas the base row retains the catalog's original judge. Each checkpoint was decoded once, and these differences carry no significance claims.

Performance versus admitted pair entries. We join round-end $\pi_{\mathbf{sec}}$ scores to cumulative Phase-1 pair entries. We count a round's assembled pairs once when its recorded $\pi_{\mathbf{sec}}$ Phase-1 loss is non-null, and zero when that update did not execute. Every one of the 450 records confirms an executed update. This measure counts entries across updates, including repeated tasks or pairs; it does not count unique annotations, epoch-weighted presentations, tokens, GRPO rollouts, oracle calls, or accelerator time. Historical records without an explicit epoch field remain missing in the audit rather than receiving a present-day default. We average cumulative counts and scores at a common round across seeds, without interpolating to a matched budget.

Observed pattern. At R10, Full's mean coverage exceeds Frozen $\pi_{\mathbf{insec}}$ by 7.4, 11.6, 8.4, 1.0, and 2.7 percentage points for Granite, Qwen 7B, Qwen 14B, Phi-4 Mini, and CodeLlama, respectively. Increased supply does not imply a uniform utility gain: at that same round, Qwen 7B has higher C&S but lower EvalPlus functionality than Frozen $\pi_{\mathbf{insec}}$; Phi-4 Mini has equal C&S and lower functionality. These fixed-round observations do not replace the selected-checkpoint results.

These diagnostics distinguish data supply from $\pi_{\mathbf{sec}}$'s measured outcomes. A larger fresh-pair pool is insufficient evidence of useful learning without improved C&S and retained functionality. The controls use their registered recipes, and Full also pays for updating $\pi_{\mathbf{insec}}$. Neither equal round labels nor equal admitted pair counts establish equal computation. The figures do not establish causal mediation, human-label efficiency, or compute efficiency, and retain the main evaluation's selection-informed status.

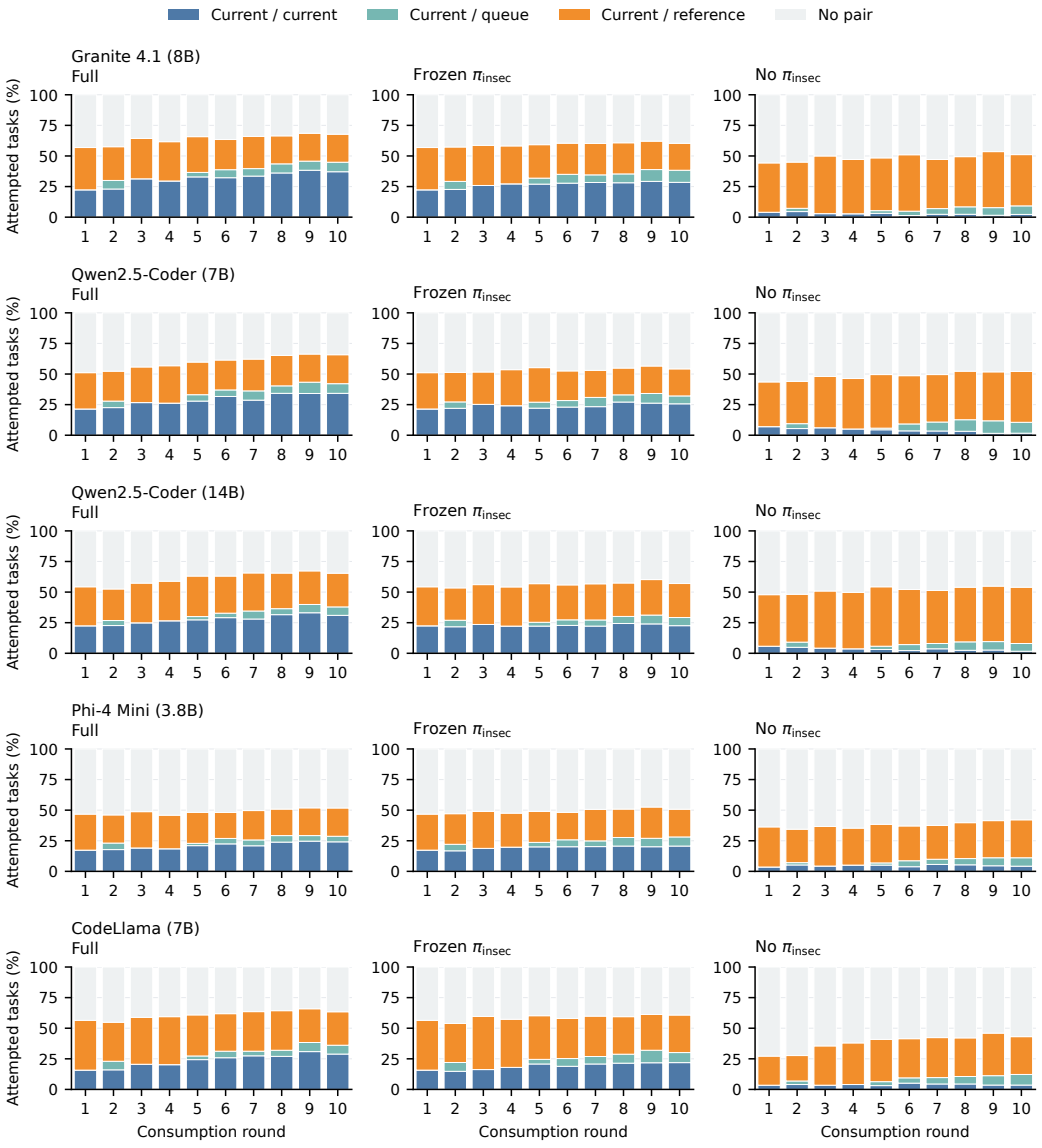

Figure 6: Pair provenance for Full, Frozen π_{insec}, and No π_{insec}. Each bar partitions attempted tasks, averaged over the three locked seeds. The colored bands sum to fresh-pair coverage; gray denotes tasks yielding no pair. "Current" refers to the consumed pool's generation pass.

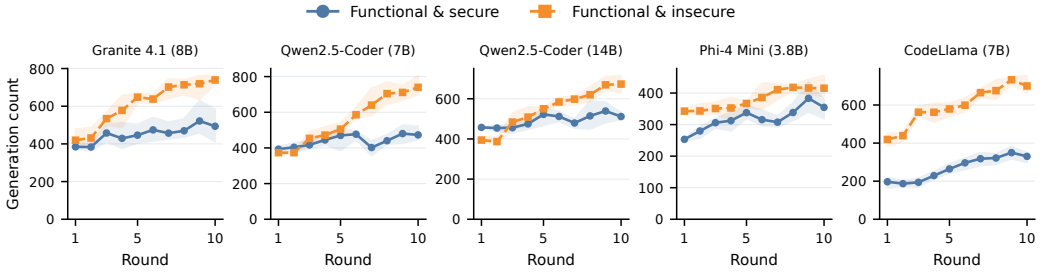

Figure 7: Full-method functional-and-secure and functional-and-insecure generation counts pooled across π_{sec} and π_{insec} in the consumed pair pool. Lines average the three locked seeds; bands span their observed range. These are counts before pair selection, not model-specific rates or post-GRPO checkpoint evaluations.

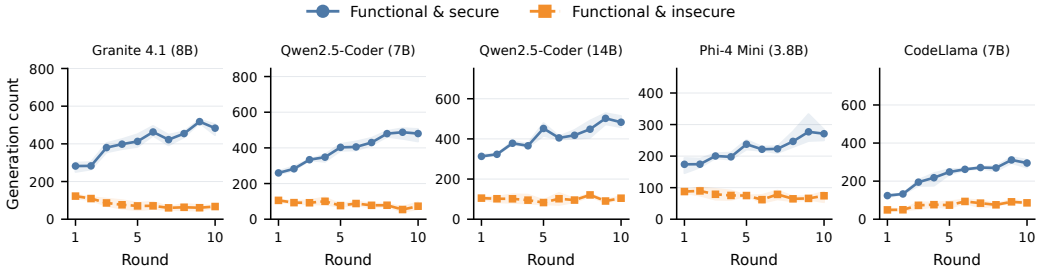

Figure 8: No π_{insec} functional-and-secure and functional-and-insecure generation counts from π_{sec} alone in the consumed pair pool, before pair selection. Lines average seeds 42/43/44; bands span their observed range. Axes match Figure 7 for each base model. Counts are not success rates or post-GRPO checkpoint evaluations.

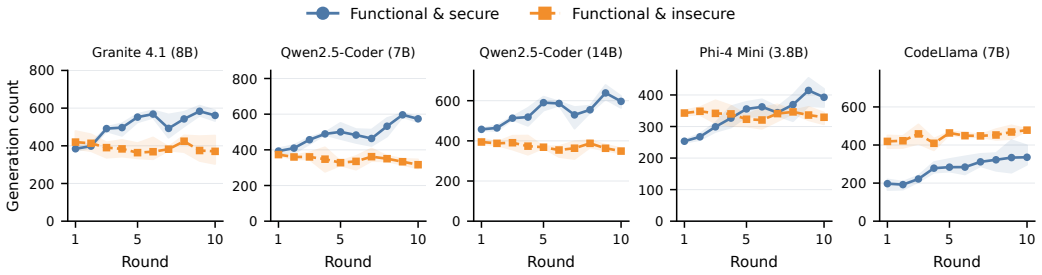

Figure 9: Frozen π_{insec} functional-and-secure and functional-and-insecure generation counts from π_{sec} and the frozen π_{insec} in the consumed pair pool, before pair selection. Lines average seeds 42/43/44; bands span their observed range. Axes match Figure 7 for each base model. Counts are not success rates or post-GRPO checkpoint evaluations.

C GENERALIZATION BEYOND REPLAY DATA

We ask whether replay improves $\pi_{\mathbf{sec}}$ on languages and CWE categories absent from its replay data. The comparison starts with Bootstrap $\pi_{\mathbf{sec}}$, immediately before replay, and ends with Ours, the selected $\pi_{\mathbf{sec}}$ checkpoint from the same training seed. Thus, $\Delta\text{C\&S} = \text{C\&S}_{\text{Ours}} - \text{C\&S}_{\text{Bootstrap}}$ measures the observed change over replay. Exposure during bootstrap is allowed: the question concerns generalization beyond replay tasks, rather than first exposure to a category.

Evaluation subsets. We define unseen categories using the exported SeCodePLT and CWEval replay corpus and use only SecCodeBench test-split tasks, because the validation split chose the selected rounds (Appendix A.4). Java is absent from the replay corpus, yielding 32 test tasks; it is present in bootstrap, with 23 pairs per selected model/seed lineage. Eight target CWEs are absent from the replay corpus: CWE-89, CWE-611, CWE-1336, CWE-917, CWE-798, CWE-328, CWE-330, and CWE-415. Four of them (CWE-611, 917, 798, and 415) are validation CWEs, so the CWE subset contains the 19 test tasks targeting CWE-89, CWE-1336, CWE-328, or CWE-330. CWE-89 occurs in bootstrap and remains eligible under this replay-based definition. Each task counts once within a subset. The subsets overlap on 12 Java tasks and are not independent evidence.

Comparison and aggregation. Table 2 matches model, seed, and evaluation task IDs across the two stages. Each stage has complete scored coverage of its subset. Rates are computed over tasks within a seed, then averaged equally over seeds 42, 43, and 44. Changes are averaged paired-seed differences, computed before rounding. The selected endpoint is R2 for Granite, R9 for Qwen 7B, R1 for Qwen 14B, R3 for Phi-4 Mini, and R7 for CodeLlama; no endpoint is selected separately for these subsets. N counts tasks per checkpoint, not across seeds.

Results and scope. Java C&S increases for Granite, Qwen 7B, and CodeLlama, by 4.17–7.29 percentage points, is unchanged for Phi-4 Mini, and decreases by 1.04 points for Qwen 14B. On replay-unseen CWEs, it increases for Granite, Qwen 7B, Phi-4 Mini, and CodeLlama, by 1.75–8.77 points, and decreases for Qwen 14B (5.26 points). Qwen 7B loses functional accuracy on both subsets (26.04 points on Java), so C&S gains do not imply uniform functionality retention. With 19–32 tasks per subset, one task changes a three-seed mean by 1.0–1.8 points. These are descriptive changes at validation-selected checkpoints. The subset definition follows the supplied replay corpus rather than an exhaustive audit of historical training inputs, and the comparison does not establish selection-independent transfer or statistical significance.