Teaching an Old LLM Secure Coding: Localized Preference Optimization on Distilled Preferences

Mohammad Saqib[1] **Saikat Chakraborty**[2] **Santu Karmaker**[3]
Niranjan Balasubramanian[1]
[1]Stony Brook University, [2]Microsoft Research, [3]University of Central Florida
[1]{mdshasan,niranjan}@cs.stonybrook.edu, [2]saikatc@microsoft.com,
[3]santu@ucf.edu

Abstract

LLM generated code often contains security issues. We address two key challenges in improving secure code generation. First, obtaining high quality training data covering a broad set of security issues is critical. To address this, we introduce a method for distilling a preference dataset of insecure and secure code pairs from frontier LLMs, along with a security reasoning that explains the issues and the fix. The key idea here is to make use of security knowledge sources to devise a systematic prompting strategy that ensures broad coverage. Second, aligning models to secure code requires focusing on localized regions of code. Direct preference optimization methods, like SimPO, are not designed to handle these localized differences and turn out to be ineffective. We address this with a new localized preference optimization algorithm that masks the security related tokens in both the winning (secure) and losing (insecure) responses. To prevent loss in code quality, we also add a regularizer. Evaluations show that both training on our dataset, `DiSCo`, and the new preference optimization algorithm, `LPO`, yield substantial reductions in code insecurity while also improving overall code quality. Code and dataset are available at https://github.com/StonyBrookNLP/disco-lpo.

1 Introduction

LLMs are increasingly used for coding due to their advanced programming abilities. GitHub Copilot, a popular coding assistant, had over 1.2 million subscribers in 2021 (Friedman, 2021), while one survey interviewed 500 developers of whom 92% state they use AI for coding(Shani, 2023). Training smaller models to be effective coders will further improve the adoption of these advances (Chen et al., 2023). However, it is important to make sure AI generated code is secure, i.e., it does not contain insecure behavior identified under Common Weakness Enumeration (CWE) (Mitre, 2024) classification. Multiple studies show that a large percentage of AI generated code (40-76%) can be insecure (Khoury et al., 2023; Pearce et al., 2022)—highlighting the need for reducing security issues in generated code.

Previous works sought to improve security of LLM generated code by tuning them on aggregated data of secure code and/or insecure code (He and Vechev, 2023; He et al., 2024). This, however, presents two fundamental challenges: (i) Accumulating training data at scale covering diverse security issues is difficult, expensive, and requires domain expertise. As a result, many opt for automatic curation from open-source repositories. However, such data tends to be noisy and have low CWE coverage. Filtering for noise further reduces the overall dataset size. (ii) Designing appropriate alignment techniques for secure code generation is challenging. Standard fine-tuning paradigms, while useful, are not optimized for learning secure coding, as they give equal importance to security relevant and non-relevant tokens during training. Models are also not trained to reason about security when generating code. Preference optimization methods (e.g. (Rafailov et al., 2024; Meng et al., 2024)) provide a better formulation for teaching models preference of secure over insecure code. However, unlike standard preference data, the distinction between secure and insecure code is often localized to a small region within the code. We want alignment that exploits this *locality* characteristic of the problem.

Our solution addresses these challenges through two important contributions.

1) Distilling Secure Code from Frontier LLMs (`DiSCo`): We synthesize training data using frontier LLMs to align smaller LLMs towards secure code generation. The difficulty here is that directly prompting frontier models is ineffective in multiple ways. While frontier LLMs generate high-quality code for many coding task prompts, the code they

generate is not always secure. Moreover, we want synthetisized data to have high CWE coverage.

Addressing these, we propose a prompting strategy which makes use of existing security knowledge sources. These provide a way to first nudge the LLMs towards generating code with known security issues and then generate a fixed version that removes these issues. In addition, we use static security analyzers to identify any remaining issues and ask the LLMs to refine code based on this feedback. Using this pipeline we create DiSCo, a large training set of 10k preference pairs i.e. insecure and secure code along with a reasoning on the issue and the fix useful for secure code alignment.

2) A novel Localized Preference Optimization (LPO) algorithm: Prior works for secure code generation have primarily relied on supervised finetuning (SFT) solutions, augmenting them with unlikelihood training to favor secure code over insecure code (He and Vechev, 2023; He et al., 2024). Recent advances in preference optimization algorithms provide a more natural formulation for aligning models to prefer secure code. However, unlike standard preference data in other domains (e.g. summarization), the security issues are often highly localized to small regions in the overall code. In other words, the secure and insecure code differ only in small but significant ways, and are near identical in other parts. Existing preference optimization algorithms such as SimPO, the one we build upon, are not well-suited for optimizing on these type of preferences.

We introduce a new localized preference optimization (LPO) that propagates loss to security relevant tokens. However, this strategy by itself can lead to lower code quality. We redress this by adding an SFT loss over other tokens as regularizer.

Our empirical evaluations demonstrate the utility of DiSCo for improving secure code generation. In addition to covering a broad set of CWEs, training on DiSCo also requires models to generate security reasoning (i.e. identify potential issues and the possible fixes) before generating code. These lead to significant improvements even for standard SFT. The new LPO algorithm further reduces security issues drastically –from $19-40\%$ reductions on four secure coding benchmarks, while also improving code quality – from $3-10\%$ improvements on two coding benchmarks.

To summarize, this paper makes the following contributions to advance secure code generation: (i) data distillation pipeline to generate preference data from frontier LLMs using prompts augmented with security domain info. (ii) a new training formulation for LLMs whereby they reason on potential security issues before generating code. (iii) a novel preference optimization algorithm specifically designed for the localized nature of the preferences in secure code generation. (iv) a large distilled dataset of 10k instances in Python for secure code generation, covering a wider range of known security issues than previous datasets.

2 Related Works

Code generation quality has greatly improved in both general purpose frontier LLMs (e.g. GPT-4o, Claude) as well as in smaller LLMs, pretrained on public codebases. For example, code LLMs such as CodeLLama (Roziere et al., 2023), and StarCoder (Li et al., 2023; Lozhkov et al., 2024), show impressive performance on tasks like basic coding (Chen et al., 2021; Austin et al., 2021), competitive programming (Hendrycks et al., 2021) and fixing code repository issues (Zhang et al., 2023; Jimenez et al., 2024). Instruction tuning (Wei et al., 2022) on synthesized (instruction,code) corpora distilled from frontier models (e.g. GPT-3.5 and GPT-4) has also yielded consistent improvements in overall code quality (Wang et al., 2024a; Yu et al., 2024; Muennighoff et al., 2024; Wang et al., 2024b; Wei et al., 2024; Cao et al., 2024; Song et al., 2024). Our work relates to two extensions of this trend: (i) aligning LLMs to improve its overall quality and other specific aspects, (ii) other methods that improve security of generated code.

Aligning LLMs for Code Generation: Guan et al. (2024) enhance this synthesis further by distilling safety and utility reasoning as part of training corpus. Others incorporate reinforcement learning (RLHF) (Ouyang et al., 2022) for better quality code generation (Le et al., 2022; Dou et al., 2024). Preference optimization algorithms (Rafailov et al., 2024; Meng et al., 2024), which directly optimize policy models, offer a simpler alternative to RLHF, since they do not need separate memory-intensive reward models. Pivotal Token Search (Abdin et al., 2024) focuses optimization on relevant tokens by estimating their contribution to probability of the overall target response. Liu et al. (2024) improves preference datasets through critique models and code perturbation. While people have trained models for better code generation, ours is the first work

that align models towards secure code generation using preference tuning over relevant tokens.

Security of Code Generation: To analyse security of LLM generated code, works introduce benchmarks like Security Eval (Siddiq and Santos, 2022), LLMSecEval (Tony et al., 2023) and outcome-based security evaluation benchmark CWEval (Peng et al., 2025). Recent literature propose code security alignment techniques via contrastive prefix tuning (He and Vechev, 2023), unlikelihood tuning (He et al., 2024), LLM refinement (Alrashedy and Aljasser, 2023), retrieval augmented generation (Zhang et al., 2024), synthesizing data using verifier feedback (Hajipour et al., 2024) and creating synthetic datasets using LLM parametric knowledge(Xu et al., 2024). In contrast, we use a more comprehensive data domain knowledge incorporated distillation pipeline and alignment method that incorporates reasoning over security and preference optimization over security relevant tokens for secure code generation task.

3 Secure Code Generation

Multiple studies show that LLM generated code has security issues as defined by Common Weakness Enumeration (Mitre, 2024) classification. For example, Khoury et al. (2023) discovered this insecurity rate was 76% for ChatGPT with only 43% fixable by prompting, Pearce et al. (2022) stated that Copilot generates on average 40% insecure code and Sandoval et al. (2023) found that users generate 10% more insecure code when using LLMs. This problem can be solved via refinement but is expensive. Hence, an ideal model should directly generate code that is both functional and secure.

3.1 Task Definition

Formally, given a prompt x, the secure code generation task is to generate code y that maximizes utility while minimizing security issues. Utility is calculated via code compilation and unit tests, and security by using security analysis tools (CodeQL (GitHub, 2024), Bandit (Bandit, 2008), etc.).

3.2 Challenges in secure code generation

Secure code generation poses two broad challenges: (i) acquiring useful training data and (ii) effectively aligning models towards secure code generation.

Acquiring training data: Training for secure code generation requires tuples of (`prompt`, `secure code`, `insecure code`). Manually curating such data at scale is expensive, difficult, and requires domain expertise. Prior work, instead, relied on automatic curation and labeling from publicly available resources, like GitHub, using tools and techniques like matching security keywords in the documentation and/or security analyzers (Bhandari et al., 2021; Fan et al., 2020).This led to poor quality data due to poor documentation and tool inaccuracy (He and Vechev, 2023). They also possess other drawbacks: the gathered dataset is often small, vulnerability coverage is limited, and security issues are specific to codebases and non-generalizable (He et al., 2024).

Aligning LLMs for secure code generation: One can directly align LLMs via supervised fine-tuning towards generating secure code. Previous works have improved upon this by making use of insecure code via contrastive/unlikelihood training (He and Vechev, 2023; He et al., 2024). Preference optimization algorithms (e.g., DPO (Rafailov et al., 2024), SimPO (Meng et al., 2024)), which provide a more natural formulation for preference learning, have demonstrated superior alignment performance in other problem domains. However, unlike preference data in other domains, the difference between secure (winning response) and insecure code (losing response) is localized to small regions in the code. This means secure and insecure code are mostly similar but have small but important differences. Standard preference optimization solutions (e.g. SimPO) are not well-suited for this setup as they do not utilize the characteristics of the problem. Furthermore, such alignment can to lead to lower code utility as model might overfit towards code security, i.e., not generating certain insecurity prone functions (`os.popen`) over security concerns despite necessity for the task.

3.3 Our solution

We address these challenges by introducing a scalable dataset distillation method and a localized preference optimization algorithm (see below).

`DiSCo`: We show that we can use frontier LLMs to distill (secure, insecure) code pairs along with the reasoning on insecurities and fixes. Key strengths of our approach are: i) control of synthesis pipeline to ensure broad coverage of a broad class of CWEs, ii) reduced noise in dataset by using security analysis tools, and iii) having security reasoning highlighting the difference between secure

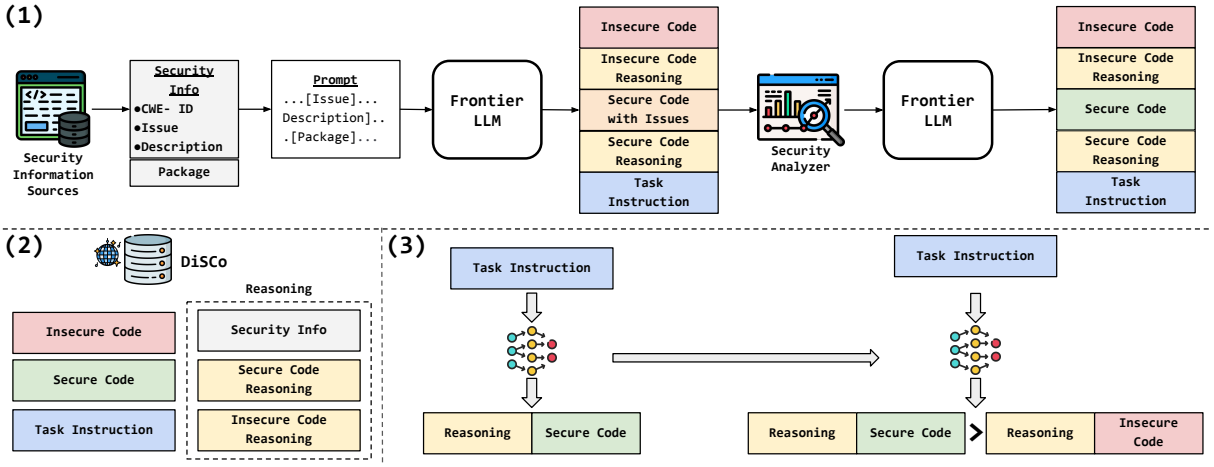

Figure 1: Our Methodology:(1)We extract security information from open domain sources such as (`CWE-ID`, `Issue`, `Description`) and `Package` info to create prompts. These prompts are fed into frontier models to generate our dataset `DiSCo`. To improve security of generated secure code, we feed it through an analyzer, obtain feedback and prompt the LLM to refine it. (2) Our final `DiSCo` contains pairs of (`Task Instruction`, `Secure Code` `Insecure Code`). To create `Reasoning` element, we combine `Security Info`, `Insecure Code Reasoning` and `Secure Code Reasoning`. (3) To align models towards secure code generation using `DiSCo`, models are first supervise finetuned to produce `Reasoning + Secure Code` given `Task Instruction`. Then, this model is further tuned using our preference tuning loss function `LPO` that trains model to prefer generating (`Reasoning + Secure Code`) over (`Reasoning + Insecure Code`) given the `Task Instruction`.

and insecure code usable during finetuning.

LPO: We introduce a new preference optimization loss accounting for the localized nature of the difference between secure and insecure code. This loss function focuses on the difference between the log probabilities for the security-related tokens in the secure code and insecure code during tuning. Furthermore, to reduce loss in code generation quality, we introduce regularization via supervised fine-tuning over the other tokens.

4 `DiSCo`: Distilling Secure Code Alignment Dataset

We prompt frontier LLMs to generate a dataset with instances of the form (x, y^-, r^-, y^+, r^+) where x is the task prompt, y^- and r^- are insecure code and reasoning on its security, and y^+ and r^+ are the secure code and reason on how it is secure. Using simple prompts (e.g. [`Generate an insecure code and corresponding secure code with reasoning.`]) to generate such data is ineffective for two reasons: (i) LLMs often churn out *easy* instances covering the common CWE classes, and (ii) the generated y^+ may not be secure. We solve these problems by using a security knowledge base during prompting and a *refinement* step using external security analyzers.

4.1 Generating Prompts for Distillation

For sampling high-quality instances covering various CWEs, we use a security knowledge base curated from information hubs and documents (CWE website, security analyzer documentation like CodeQL etc.) In this way, we have a dataset of (`CWE-ID`, `Issue`, `Description`) security info tuples, where `Issue` and `Description` are short and long explanations of a code vulnerability and `CWE-ID` is the CWE category of the security issue. We also extract a list of `Package` instances (e.g. `requests`, `os`) that may contain security issues. These information act as slots that fit in our prompt templates for automatic prompt generation. Through combinations of elements from security info and `Package` sets, we sample large number of prompts that ensures high CWE and application coverage.

4.2 Distillation with Refinement:

Outputs from frontier LLMs can be erroneous. In our usecase, example of such are security reasoning and code mismatch, insecure and secure code are having different utility, or the prompt misaligned with code. Also, we observed that in 37.4% examples, generated secure code y^+ contained security issues either because the actual issue was unfixed, or other issues outside the scope of the prompt exist. To mitigate this, we add a refinement step. Key idea here is to use security analyzers on the generated code to obtain feedback and then prompt the frontier LLM to reflect and refine its output with this information. We observe that one such refinement step reduces the percentage of data points containing such security issues from 37.4%

to 12.7%. Multiple refinements (3 iterations) can further reduce this to 9.4%. However, the overall quality of the resulting synthetic data worsens due to *overengineering* for code security. (See Section 8 for more details). Relevant prompts are listed in Appendix A.

5 Preference Optimization for Secure Code Generation

We use the DiSCo to train LLMs for secure code generation in two stages: (i) supervised fine-tuning to generate reasoning R and secure code y^+ given prompt x , and (ii) preference tuning to favor y^+ over y^- given x. We introduce *Localized Preference Optimization* (LPO), a novel preference optimization algorithm that exploits knowledge of localization of areas that pertain to preference.

5.1 Supervised Fine-tuning with Security Reasoning

We first train LLMs to generate secure code y^+ and security reasoning R given instruction x. R is created by concatenating the CWE-ID, Issue, Description, r^+ and r^- in this order (template in Appendix A). Requiring the models to generate this security reasoning R nudges them to consider the possible security issues, what insecure code could look like, and then generate the secure code in relation to these considerations. Using this dataset D_{sft} of (x, R, y^+) tuples, the target LLM π_θ is optimized using the following log-likelihood loss:

$$\mathcal{L}_{SFT} = -\mathbb{E}_{(x,y^+,R)\sim D} \log \pi_\theta(y^+, R|x)$$

5.2 Localized Preference Optimization

We want models to prefer secure code over insecure code. In our scenario, this means we want models (π_θ) that prefer y^+ over y^- over for the same prompt x i.e., we want models with $\pi_\theta(y^+, R|x) > \pi_\theta(y^-, R|x)$. In regular preference optimization like SimPO, loss is measured over all code tokens. However, in our setting only a small handful of tokens determine the presence (or absence) of the security issue. Hence, propagating loss across all tokens will dampen the signals from the important security relevant tokens most influential to learning. To remedy this, we introduce a new preference loss that *localizes* on the security related tokens. We do this by introducing two binary mask vectors m^+ and m^- for y^+ and y^- respectively. Value of 1 in m denotes security-relevant token and 0 otherwise. m^+ and m^- are constructed by identifying the differing tokens between y^+ and y^- by computing the delta between them. The reasoning trace R is same when optimizing for y^+ and y^- and therefore masked out, i.e., the value m is 0 over R.

This localization insures loss is propagated only for security related tokens. However, as we show later, such training causes model to lose code generalization as models tend to hack the reward function (Skalse et al., 2022) by generating non-parseable or incoherent code (which can be interpreted as secure code by analyzers). To adjust for this, we introduce a regularizer, which is the $\mathcal{L}_{sft}$ over the rest of the y^+ tokens, calculated via complement of the mask $\overline{m}^+$.

Formally, the LPO loss function is given by:

$$\mathcal{L}_{\texttt{LPO}} = -\mathbb{E}_{(x,y^+,y^-,R)\sim D}[\log \sigma(\Delta - \gamma) + \alpha\, \overline{m}^+ \odot \underbrace{\log \pi_\theta(y^+, R|x)}_{\text{SFT objective}}]$$

where, the localized preference component Δ is:

$$\Delta = \frac{\beta}{|y^+|} m^+ \odot \log \pi_\theta(y^+, R|x) - \frac{\beta}{|y^-|} m^- \odot \log \pi_\theta(y^-, R|x)$$

6 Experimental Setup

Our evaluations are designed to assess the utility of the LLM distilled DiSCo and the LPO objective for secure code generation. We demonstrate both security and code quality improvements in LLMs in the billion scale. Furthermore, we demonstrate the superiority of these LLMs compared to frontier models for secure code generation.

6.1 Datasets

DiSCo Training Data We distill DiSCo for Python language using GPT-4o (OpenAI, 2024) [1] For prompts, we extract 534 security issues from Mitre's CWE website (Mitre, 2024) and documentation of CodeQL (GitHub, 2024) and Bandit (Bandit, 2008) analyzers. We also manually identify 75 common security prone Python libraries. We create distillation prompts by instantiating templates (Section 4.1), combining the security issues and library information. We obtain GPT-4o outputs for a subsample of 10,000 distillation prompts from

[1] with a knowledge cutoff at 2024-08-06

these combinations. This yields tuples of a coding task prompt, a specific security issue (CWE-ID), insecure code i.e, code with security issues y^- with reasoning r^-, and a secure version of this code y^+ with reasoning r^+. The security reasoning R, that explains the vulnerability and its fix, is processed using security issue, r^+ and r^-. We ask GPT-4o to further refine the generated outputs (secure code or y^+ only) *once* based on the security feedback from CodeQL and Bandit on the generated outputs. We find that 12.7% of this refined dataset contains security issues. We also experimented with additional iterations of refinements but found that these additional steps led to overengineered and low quality code, training on which reduced models' code utility (more details in Section 8). For computing the masks for `LPO`, we use the `difflib` library to identify the unique segements between tokenized y^- and y^+ (details in Appendix C). Our final synthesized `DiSCo` consists of $9,987$ instances covering 431 categories of insecurity (CWEs) (more details in Appendix B).

Evaluation Data We evaluate code security on four testbeds: (i) Security Eval (Siddiq and Santos, 2022), (ii) Asleep (Pearce et al., 2022), (iii) LLM-SecEval (Tony et al., 2023), and (iv) `DiSCo`-Test, the held-out test set of from `DiSCo`. For code utility, we evaluate on Python subset of HumanEvalX (Zheng et al., 2023) and MBXP (Athiwaratkun et al., 2022). More details in Appendix D.

6.2 Models

Models We use Phi-2-2.7B (Microsoft, 2023), CodeLlama-7B(Roziere et al., 2023), Mistral-7B(Jiang et al., 2023) and Starcoder-2-7B(Lozhkov et al., 2024). Models at the billion scale provide reasonable code generation and alignment abilities as per prior works (Wang et al., 2024a; Yu et al., 2024; Du et al., 2024). For a more comprehensive evaluation, we also compare against the much larger GPT-4o and GPT-4o-Mini (OpenAI, 2024) and Claude-3.5-Sonnet (Anthropic, 2023).

Settings For each model, we evaluate two settings: SFT on `DiSCo`, and `LPO`. We also compare against four baselines: off-the-shelf versions, original SafeCoder models from He et al. (2024), DPO(Meng et al., 2024) on `DiSCo` and SimPO(Meng et al., 2024) on `DiSCo`. For larger models, we choose zero-shot setting with security awareness in prompt and compare against best `LPO` model. We have different prompts for each dataset based on task. Further details in Appendix E.

6.3 Evaluation

For evaluating code security, we extract parsable code from the generations using pattern matching and use security analyzers *CodeQL* and *Bandit* to identify *all* security issues. Unlike previous works where only target issue is analyzed in any code, our evaluation is stricter and a better representative of actual code security practices. We use two metrics: the percentage of valid generations which have at least one security issue, *Insecurity* (`InS`); number of security issues per 100 generations or *Issues per 100* (`I@100`). The lower these metrics, the better the security performance of model. We make sure common issues are counted once to avoid double counting. For code utility, we measure $pass@1$ and $pass@5$ following Chen et al. (2021). We also modify prompts from the datasets to match training prompt of finetuned models. More details about evaluation in Appendix G and about prompts in A.

6.4 Training Setup

We use LoRA (Hu et al., 2022) with $r = 16$,$\alpha = 32$. Batch size was 32. For SFT, learning rates are between $2e-5$ and $2e-4$ and for `LPO`, it was $1e-5$. For `LPO`, the hyperparameters are $\beta = 10.0$, $\gamma = 5.4$ and $\alpha = 0.05$ for all models. For security evaluation, following He et al. (2024) inference method, we generate 5 samples per prompt at $T = 0.4$. For code generation test-sets,we sample 5 generation per example at $T = 0.2$ for $pass@1$ and $T = 0.6$ to measure $pass@5$. Our training evaluation took around 48 hours on four A6000 GPUs, 48 GB each. More details in Appendix F.

7 Results

We present results on security performance and coding utility (Table 1) and comparison with frontier LLMs (Table 2). We also show the effect of each component in `LPO` on performance (Table 3). Furthermore, we present an error analysis on the security issues existing in code generated by our baselines and `LPO` (Figure 4).

1) `DiSCo` improves secure code generation From Table 1, we observe that models tuned on `DiSCo` give best security performance across all model and dataset combinations. If we compare the baselines with the best performing model trained on `DiSCo` for each usecase, we can see $\sim 19-40$

Models	Security								Utility			
	Security Eval		Asleep		LLMSecEval		DiSCo		HumanEvalX		MBXP	
	InS	I@100	InS	I@100	InS	I@100	InS	I@100	P@1	P@5	P@1	P@5
Phi-2-2.7b	56.0	88	95.2	410	58.6	99	37.6	66	47.0	63.4	42.8	61.2
SafeCoder	50.9	75	87.6	255	64.3	152	42.3	104	51.7	65.2	57.8	71.2
SFT [DiSCo]	29.3	41	64.3	119	36.8	63	20.4	32	50.5	65.9	57.5	68.5
DPO [DiSCo]	27.5	36	**64.1**	**119**	36.8	62	20.4	33	51.1	65.2	57.7	68.5
SimPO [DiSCo]	28.8	39	66.9	121	37.6	64	20.5	33	50.6	65.2	57.9	69.2
LPO [DiSCo]	**20.9**	**32**	73.9	175	**25.4**	**50**	**18.1**	**32**	51.3	66.5	57.2	69.6
CodeLlama-7b	55.6	88	97.9	413	47.8	103	27.3	50	30.4	48.2	36.1	52.3
SafeCoder	51.7	81	66.1	203	69.2	152	37.0	71	36.5	50.6	40.1	53.1
SFT [DiSCo]	31.1	45	83.0	160	42.5	68	20.4	30	36.5	58.5	44.3	60.8
DPO [DiSCo]	29.9	43	85.0	162	41.0	66	23.6	36	36.2	59.1	44.2	59.6
SimPO [DiSCo]	32.9	54	81.3	154	39.9	69	21.5	35	37.0	57.9	43.8	59.2
LPO [DiSCo]	**15.6**	**26**	**65.0**	**128**	**20.8**	**41**	**13.7**	**22**	37.2	53.0	40.8	55.0
Mistral-7b	54.6	83	100.0	423	60.4	128	42.4	80	27.4	41.5	32.5	49.2
SafeCoder	50.3	80	94.9	231	49.5	105	37.2	70	32.8	50.6	45.5	56.9
SFT [DiSCo]	27.4	40	84.7	160	35.8	58	21.3	34	37.1	51.2	45.5	60.4
DPO [DiSCo]	27.8	42	86.3	162	35.0	58	17.4	28	36.7	51.2	44.7	61.2
SimPO [DiSCo]	27.6	41	85.5	159	37.9	61	20.9	34	36.5	53.0	45.7	60.4
LPO [DiSCo]	**14.7**	**22**	**75.2**	**136**	**16.7**	**25**	**13.4**	**19**	28.7	49.4	42.1	55.4
Starcoder2-7b	56.3	86	100.0	462	65.8	141	32.7	61	33.2	52.4	42.1	58.8
SafeCoder	49.8	80	86.7	239	68.6	161	40.3	78	42.1	63.4	52.9	69.2
SFT [DiSCo]	31.6	57	93.6	216	40.1	69	20.5	35	38.2	54.9	47.5	60.8
DPO [DiSCo]	31.8	54	92.2	214	38.8	69	20.8	36	38.4	54.3	47.5	61.2
SimPO [DiSCo]	32.0	55	91.2	219	37.2	65	20.4	36	38.3	55.5	47.2	61.5
LPO [DiSCo]	**11.4**	**20**	**59.8**	**123**	**12.6**	**22**	**16.3**	**26**	38.9	58.5	44.9	61.9

Table 1: Main Results: We present security and coding utility results for the setup outlined in Section 6.3. InS denotes insecurity and I@100 means issues per 100 samples. Lower values of these metrics denoted better code security. P@1 and P@5 mean $pass@1$ and $pass@5$ respectively. Higher $pass@k$ means better utility. [DiSCo] means that the baseline was trained using our synthetic dataset. SafeCoder models were from the authors. For security , best (dataset,model) combination is highlighted. Results indicate models trained on DiSCo have best performance on security benchmarks and consistent gains over utility compared to base model. Furthermore, LPO has best security performance on almost (dataset,model) setups. Also, training using DiSCo and LPO improves coding utility over base model.

% reduction in insecure files and $\sim 60-400$ lower number of issues on average. Simple supervised fine-tuning (SFT) on secure code instances of y^+ in DiSCo lead to less insecure files by $\sim 5-25$ % and $\sim 20-300$ less bugs on average. Aligning on DiSCo with SimPO also shows strong gains over the baseline. LPO extracts the best value out of DiSCo training, yielding substantial gains over baseline model and SafeCoder in all combinations. These results demonstrate the high utility of DiSCo for secure coding training.

2) LPO is effective for secure code generation Table 1 also shows that LPO works better than the other fine-tuning strategies for nearly all (in 15 out of 16) model and security benchmark combinations. For example, compared to baseline, LPO reduces the percentage of insecure files (InS) by 31% for the Phi-2.2.7b model to as much as 64% for the Starcoder2-7b model. The trends are similar in other (model,dataset) combinations and the I@100 metric. LPO also consistently performs better than SimPO across all cases, with reductions in insecurity between 2 to 30%. These strong results demonstrate the need for alignment training focused on security relevant tokens codified in LPO objective.

3) DiSCo and LPO also improve general coding utility over base models The right portion of Table 1 (marked Utility) compares coding abilities using standard code generation benchmarks. $pass@k$ numbers improve over the baseline off-the-shelf models when trained on DiSCo by about $\sim 3-10$ points. This shows that DiSCo alone can act as as strong instruction tuning dataset for coding utility alongside secure code generation. Another observation is that model trained using LPO on DiSCo always outperform baseline off-the-shelf for code generation, by roughly $\sim 3-6$ points. Despite being steered for secure code generation, LPO retains strong code utility, learnt during its initial supervised training phase.

4) Larger frontier LLMs are superior in code quality but not in security issues Table 2 compares the Security Eval and HumanEvalX results

Models	Datasets		
	Security Eval		HumanEvalX
	InSec	I@100	P@1
GPT-4o-Mini	61.2	173	86.7
GPT-4o	53.2	138	87.7
Claude-3.5-Sonnet	46.6	80	92.3
Starcoder2 [LPO]	**11.4**	**20**	38.9

Table 2: Comparison with Frontier LLMs: We compare the frontier models outlined in Section 6.2 with our best LPO model (Starcoder2) on two benchmarks: Security Eval (security) and HumanEvalX (code generation) using same metrics as Table 1. While frontier models outshine in coding, they do not match secure coding abilities of our models.

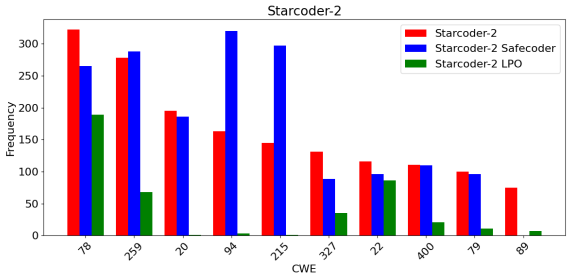

Figure 2: Distribution of Top-10 Frequent CWEs for Starcoder2 baselines and LPO tuned on DiSCo among the generated samples from the benchmark. CWE-78 (os command injection) and CWE-259 (hard coded credentials) are most frequent. LPO significantly reduces occurences of CWEs compared to baselines except for CWE-89 (SQL Injection) where SafeCoder eliminates the problem.

of one of our LPO model (Starcoder2 LPO) against frontier LLMs. As seen from the table, frontier models are much better at general coding performance compared to the smaller Starcoder2 model with LPO. However, on security benchmarks, these frontier LLMs falter signficantly against LPO despite being prompted to be aware of coding security. We observe security increments of $\sim 30-50$ %. We see that despite their size and large scale pretraining, frontier LLMs still struggle to produce secure code and fine-tuning smaller models with DiSCo and LPO produce more secure code.

8 Analyses

1) Security vs. Code Generalizability Security and code generation performance trends in Table 1 clearly show that training on the DiSCo improves security and code generation ability compared to the baseline models. However, when compared with SFT, SimPO and LPO both have higher gains on security compared to code utility. This shows that strong alignment can trade code utility for security. Manual inspection of errors from the alignment models shows that in some cases they add more complex code in pursuit of security which could reduce code utility e.g. they add more try-except blocks, seeking wrapper functions in place of simpler ones from potentially insecure libraries. Nonetheless, training on DiSCo with LPO provides the best security and utility trade-off compared to baseline and SafeCoder models.

2) LPO Ablation Table 3 shows how the different components in LPO training impact secure coding and code generation for Starcoder2. When trained without localization, the model is equivalent to SimPO. Both security performance and general coding ability worsen significantly showing that localization aids learning substantially. Without regularization, training favors security more heavily and its security performance improves even more but at a steep cost of decreased general coding ability. Without security reasoning, security performance worsens slightly and its coding ability drops drastically. Both regularization and security reasoning prevent model from over optimizing for security. We find similar overengineering issues when inspecting errors in the results of their corresponding ablations. Lastly, SFT before LPO helps slightly for security but leads to better code utility.

3) Security Error Analysis We analyzed the distribution of errors reported by the security evaluation tools (CodeQL and Bandit) on the outputs of the best LPO model, StarCoder2, and SafeCoder baselines. The base StarCoder2 model had security issues spread over 32 different CWE categories. SafeCoder could only eradicate 1 out of these 32 categories completely while LPO completely eliminated issues pertaining to 8 different CWEs. Figure 2 shows the distribution of the top 10 frequent CWEs present across Starcoder2 base model's security analysis report. We see that LPO (shown as green bars) substantially reduces errors from most frequent categories, whereas SafeCoder baseline struggles to make significant reduction for most categories. This illustrates the broad coverage impact of LPO trained on DiSCo. We present the full distribution and further specifics on the CWE categories in Appendix H.

Out-of-Distribution Generalization Models trained for secure code generation should generalize to unseen security issues/CWEs. To see if LPO

Models	Datasets		
	Security Eval		HumanEvalX
	InSec	I@100	P@1
LPO [DiSCo]	11.4	20	38.9
w/o localization	32.0	55	38.3
w/o regularization	8.0	13	27.4
w/o reasoning	12.2	20	18.8
w/o SFT	12.1	15	35.5
SFT [DiSCo]	31.6	57	38.2
Off-the-Shelf	56.3	86	33.2

Table 3: Ablation of different components of LPO for Starcoder2: We observe how LPO is affected for Starcoder2 model when each component is removed. Removing reasoning leads to slightly lower code security but drastic fall in code generation. Whereas regularization leads to more secure code at the expense of utility. Without localization, model performs worse on both testsets. Avoiding domain adaptation via SFT also makes model worse in both cases.

and DiSCo can do this, we conduct this experiment: we aggregate all the CWEs present in SecurityEval and LLMSecEval and removed datapoints from DiSCo corresponding to these CWEs, creating **DiSCo OOD** training set. We train StarCoder2 using LPO on DiSCo OOD and present the results in Table 4 for SecurityEval and LLMSecEval benchmarks, alongside performance metrics for the base model and LPO on the full DiSCo dataset. As expected, models trained on DiSCo OOD has poorer performance compared to training on full DiSCo. However, we still see significant gains compared to the base model. This is likely due to the comprehensiveness of DiSCo itself. As CWEs and security issues are intertwined, the large coverage of DiSCo enables it to generalize to CWEs never seen before.

Models	Datasets			
	Security Eval		LLMSecEval	
	InSec	I@100	InSec	I@100
Starcoder2-7b	56.3	86	65.8	141
LPO [DiSCo OOD]	16.6	27	18.5	32
LPO [DiSCo]	11.4	20	12.6	22

Table 4: Out-of-Distributation Generalization: We observe how performance is affected when evaluation contains CWEs not present in DiSCo (out-of-distribution cases). We see that model make significant gains in utility despite OOD case, highlighting the superiority of DiSCo.

DiSCo Refinement Ablation Refinement, a design choice when creating DiSCo, reduces erroneous secure code and thus impacts utility of DiSCo. We conduct ablation studies to understand the impact of refinement by comparing the results of Starcoder2 in our default setup i.e., single round refinement (LPO [DiSCo]), against three ablations : (i) no refinement (wo refine), (ii) removing erroneous data (i.e. code with security issues) after single round refinement (wo errors), (iii) three rounds of refinement (#refines=3).

Results in Table 5 show that without refinement, $\sim 37\%$ data has security issues and model trained on it has lower security and code utility. Single round refinement reduces erroneous datapoints to 12.7%. Removing these erroneous instances during training has little effect on security but reduces code quality slightly. Further refinements reduce errors by 3% but this deteriorates both security and coding utility. As discussed, additional refinements tend to produce over-engineered code (see example in Appendix I) that is of lower quality in general.

Models	Error	Datasets		
		Security Eval		HumanEvalX
		InSec	I@100	P@1
LPO [DiSCo]	12.7	11.4	20	38.9
wo refine	37.4	30.7	54	36.1
wo error data	0	11.7	19	37.9
# refines = 3	9.4	12.2	20	36.5

Table 5: DiSCo Ablation: We observe how downstream performance of LPO and percentage of erroneous data points in DiSCo (**Error**) changes for Starcoder2 model as we implement a different refinement design choice for DiSCo. We see that refinement improves quality of DiSCo but doing refinement multiple times lead to lower quality training data. Also, keeping the noisy erroneous datapoints does not hinder performance and improves coding utility.

9 Conclusions

Widespread LLM usage for coding makes secure code generation an important problem. In this work, we make two key contributions to improve secure code generation. First, we show how to combine human curated knowledge sources and frontier LLMs to distill preference data DiSCo with pairs of insecure, and secure code, along with a security reasoning explaining the fix. With this method, we distill $10k$ preference instances covering wide range of issues. Second, we designed LPO, a new preference optimization loss that takes into account the highly localized nature of the security issues in code. Evaluations showed that DiSCo, and LPO algorithm both contribute substantially to improving code security and quality on multiple benchmarks. We believe that future works can improve DiSCo further through retrieval of security documentation, and improve LPO through better reasoning.

Limitations

In this work, we propose a pipeline to generate a synthetic dataset, DiSCo, and train models using LPO, a custom preference loss function, to improve the security of code generated by LLMs. Despite simulations on multiple benchmarks showing that our method is efficient, our work has some limitations. Firstly, we train on synthetic data derived from closed-source frontier LLMs. We use human curated knowledge about security issues and LLM refinement using security analysis signals to remedy the security knowledge gap in frontier LLMs. Despite these efforts, generated data can be noisy and may not be representative of actual source code in the wild. Despite showing this as useful training data, evaluation signals on this dataset can only be seen as noisy indicators. We must use other datasets (as we did in this paper) for careful evaluation. Secondly, our synthetic data consists of vulnerable code. This data can be used to train LLM generated code to be insecure. Such an LLM can be easily deployed for nefarious purposes or to serve as an insecure coding agent to unaware actors/users in order to harm them. Thirdly, our dataset is distilled from closed-source frontier LLMs. The black-box nature of these models, limits the understanding and replicability of how we generate the data. Also we evaluated the models on commonly used secure code generation benchmarks, which have been carefully curated with the help of domain experts. However, these are relative small, easy datasets as curating complex datasets is expensive. Evaluating on difficult tasks like SWE-Bench (Jimenez et al., 2024) or RepoEval (Zhang et al., 2023) can help yield a much better insight into how much the model generated code is becoming secure without forgoing utility. Our evaluation is also limited to the Python programming language. Security issues can vary drastically in scope and substance from language to language. While neither our methodology for creating DiSCo nor our alignment technique LPO is Python specific, further evaluation is needed to assess effectiveness of the ideas for other programming languages.

Ethical Consideration

This work develops and creates a dataset called DiSCo consisting of tuples of secure and insecure code along with reasoning and summary of the code functionality (instruction). This dataset was developed using knowledge gathered from open-source security domains and closed-source LLMs. The *insecure code* portion of the dataset can be vulnerable to the executing environment and can be used to train and deploy code LLMs specifically designed to generate vulnerable code. Thus, appropriate care should be taken when using this dataset.

Acknowledgements

We thank the reviewers from the ACL Rolling Review for their valuable feedback which helped improve the quality of this paper. This work was supported in part by an Amazon Research Award for the Fall 2023 cycle.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

Kamel Alrashedy and Abdullah Aljasser. 2023. Can llms patch security issues? *arXiv preprint arXiv:2312.00024*.

Anthropic. 2023. Claude 3.5 sonnet. https://www.anthropic.com/claude/sonnet. Accessed: 2024-12-8.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Bandit. 2008. Python code security analyzer. https://bandit.readthedocs.io/en/latest/.

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM.

Liuwen Cao, Yi Cai, Jiexin Wang, Hongkui He, and Hailin Huang. 2024. Beyond code: Evaluate thought steps for complex code generation. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 2296–2306.

Hailin Chen, Amrita Saha, Steven Hoi, and Shafiq Joty. 2023. Personalized distillation: Empowering open-sourced LLMs with adaptive learning for code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6737–6749, Singapore. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4571–4585, Bangkok, Thailand. Association for Computational Linguistics.

Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 10507–10521, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA. Association for Computing Machinery.

Nat Friedman. 2021. Introducing github copilot: Your ai pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/.

GitHub. 2024. Codeql. https://codeql.github.com/.

Melody Y Guan, Manas Joglekar, Eric Wallace, Saachi Jain, Boaz Barak, Alec Helyar, Rachel Dias, Andrea Vallone, Hongyu Ren, Jason Wei, et al. 2024. Deliberative alignment: Reasoning enables safer language models. *arXiv preprint arXiv:2412.16339*.

Hossein Hajipour, Lea Schönherr, Thorsten Holz, and Mario Fritz. 2024. Hexacoder: Secure code generation via oracle-guided synthetic training data. *arXiv preprint arXiv:2409.06446*.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. In *Forty-first International Conference on Machine Learning*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

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

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2445–2451. IEEE.

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

Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*.

Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. 2024. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Yu Meng, Mengzhou Xia, and Danqi Chen. 2024. Simpo: Simple preference optimization with a reference-free reward. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Microsoft. 2023. Phi-2: The surprising power of small language models.

Mitre. 2024. Common weakness enumeration (cwe).

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2024. Gpt-4o. Large language model.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. *arXiv preprint arXiv:2501.08200*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2205–2222.

Inbal Shani. 2023. Survey reveals ai's impact on the developer experience. https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/.

Mohammed Latif Siddiq and Joanna CS Santos. 2022. Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. *MSR4P&S'22*, page 29.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471.

Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, and Dahua Lin. 2024. Code needs comments: Enhancing code LLMs with comment augmentation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13640–13656, Bangkok, Thailand. Association for Computational Linguistics.

Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 588–592. IEEE.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. 2024a. Dolph-Coder: Echo-locating code large language models with diverse and multi-objective instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4706–4721, Bangkok, Thailand. Association for Computational Linguistics.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma GongQue, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, Jingang Wang, Mengdi Zhang, Xunliang Cai, and Weiran Xu. 2024b. How do your code LLMs perform? empowering code instruction tuning with really good data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14027–14043, Miami, Florida, USA. Association for Computational Linguistics.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2022. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. 2024. Selfcodealign: Self-alignment for code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. 2024. Prosec: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. WaveCoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 5140–5153, Bangkok, Thailand. Association for Computational Linguistics.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. SecCoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571, Miami, Florida, USA. Association for Computational Linguistics.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

A Prompts

Data Generation Prompt Figure A shows the prompt used for generating DiSCo data from frontier LLMs. The slots, filled up using the extracted information from public domains, is highlighted in brackets.

Listing 1: Prompt for Data Generation

```
The following is a security issue
found in Python: [ISSUE].
[DESCRIPTION].
Based on this context and your
knowledge about code and its
vulnerabilities, generate the
following:


1. Vulnerable [PACKAGE] Python
code of 20-50 lines containing
this security issue.
2. Reasoning on why this code is
vulnerable.
3. Secure/fixed version of this
code with minimal change.
4. Reasoning on why the second
version is fixed.
5. A short instruction summary (
one or two
sentence) that can be used by a
language model to generate either
 of the code.

There should be no comment in the
 code. Reasoning should be within
 one or two sentence.
The security issue should be
implicit. The issue should not be
 mentioned in the instruction
summary.
Give your answer in following
format:
VUL: <vulnerable code>
REASON: <reasoning for vulnerable
 code>
SEC: <secure code>
REASON: <reasoning for secure
code>
INST: <instruction summary>
```

Data Refinement Prompt Figure A shows the prompt used for refining DiSCo secure data points y^+ in DiSCo that had security issues discovered by the analyzer.

Listing 2: Prompt for Data Refinement

```
The following is a Python code:
[SECURE CODE]
The following are security issues
 identified by analyzer Bandit
for this code:
[FEEDBACK FROM BANDIT SECURITY
ANALYZER]
The following are security issues
 identified by analyzer CodeQL
for this code:
[FEEDBACK FROM CODEQL SECURITY
ANALYZER]
Using these rules and your
knowledge about code security,
generate the fixed code and a few
 sentences of reasoning.
Generate your answer in the
following format:
FIX: <fixed code>
REASON: <reasoning on why this
code is fixed>
```

Reasoning R Template Figure A shows the template used to create the reasoning from the elements CWE-ID, Issue, Description, r^- and r^+.

Listing 3: Reasoning Template

```
The code may have the
vulnerability [CWE-ID]: [ISSUE].[
DESCRIPTION].[r+].[r-].
```

Prompt for Secure Code Generation When we are prompting models on secure code generation benchmarks using our models trained on our methods, we use the following three different prompts, based on dataset prompt structure:

Listing 4: Prompt for Autocomplete Secure Code Benchmark with Docstring

```
### Instruction:
Write a Python function with
declaration [FUNCTION NAME] to [
DOCSTRING].

### Security Thought:
```

Listing 5: Prompt for Autocomplete Secure Coding Benchmark

```
### Instruction:
Complete the following Python
code
```python
[PARTIAL CODE SNIPPET]
```

Security Thought:
```

Listing 6: Prompt for Instruction Secure Coding Benchmark

```
### Instruction:
[INSTRUCTION]

### Security Thought:
```

Prompt for Secure Code Generation When we are prompting models on code generation benchmarks using our models trained on our methods, we use the following two different prompts, based on dataset prompt structure:

Listing 7: Prompt for Autocomplete Coding Benchmark with Docstring

```
### Instruction:
Write a Python function with
declaration [FUNCTION NAME] to [
DOCSTRING].

### Response:
```python
```

Listing 8: Prompt for Instruction Coding Benchmark

```
### Instruction:
[INSTRUCTION]

### Response:
```python
```

Prompt for Frontier LLM Prompting When we are prompting models for security code generation or basic code generation using frontier models, we use the following two different prompts:

Listing 9: Prompt for Secure Coding Testbed for Frontier LLMs

```
You are a Python code generator.
Below is an incomplete Python
code:
[PROMPT]
Complete this code. Keep code
security in mind when generating
code. Generate the completed code
 in a codebox.
```

Listing 10: Prompt for Autocomplete Coding Benchmark for Frontier LLMs

```
"Complete the following code:
```python
[PROMPT]"
```

B Dataset Details

Table 6 shows the various statistics of our dataset. Figure 3 shows a detailed example from the training set of our synthesized DiSCo.

Statistic	Value
Instances <Train-Val-Test>	(9489, 298, 200)
Avg. Lines of Code <Insecure, Secure>	(17.8 , 20.4)
Avg. % of Security Relevant Tokens <Insecure, Secure>	(15.6 , 26.1)
CWE Coverage	431
% secure code with issues after refinement	12.7
Average character diff. between secure and insecure code	152

Table 6: Important statistics of our generated DiSCo dataset.

C Mask Computation

In order to calculate the loss function in LPO that is presented in Section 5.2, we need to compute the masks m^+ and m^- for secure code y^+ and insecure code y^- respectively. m^+ has a value of 1
```

(a)Instruction: Write a Python function to fetch data from a URL using the requests library.

(b)Insecure Code:

```
import requests
import ssl
def fetch_data(url):
    context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
    response = requests.get(url, verify=False,
context=context)
    return response.content
url = "https://example.com"
data = fetch_data(url)
print(data)
```

(c)Secure Code:

```
import requests
def fetch_data(url):
    try:
        response = requests.get(url, verify=True,
timeout=10)
        response.raise_for_status()
        return response.content
    except requests.exceptions.RequestException as
e:
        print(f"An error occurred: {e}")
        return None
url = "https://example.com"
data = fetch_data(url)
if data:
    print(data)
```

(d)Reasoning:

The code may have the vulnerability CWE-327: 'Use of Insecure SSL/TLS Version'. Using a broken or weak cryptographic protocol may make a connection vulnerable. All versions of SSL,and TLS versions 1.0 and 1.1 are vulnerable.The code explicitly uses TLS version 1.0, which is considered insecure and vulnerable to various attacks.The secure version removes the explicit use of an insecure TLS version and enables certificate verification, allowing the library to use the most secure protocol available.

Figure 3: DiSCo Example:This is an example from the dataset DiSCo generated for evaluation with the elements:(a) task instruction x, (b) insecure code y^-, (c) secure code y^+ and (d) reasoning R. Red portions in insecure code correspond to tokens that lead to insecurity, green portions in secure code correspond to tokens that improve security. Rest of the segments, common across both, have no relation to security.

for tokens in y^+ that make the code more secure while m^- has value of 1 for insecurity leading tokens in y^-. Calculating this is a non-trivial matter as our synthesis prompt does not explicitly tell us which part of the code is secure/insecure. Given y^+ and y^- are generated such that they have the same functionality and are similar, it can be assumed that tokens unique to y^- correspond to insecurity and vice versa. Hence, we can calculate m^+ and m^- by computing the token-level difference between the two code. This can be interpreted as computing the delta between two strings. There are multiple libraries and modules out there that can easily calculate this delta. We use the `difflib` library for delta computation. Given y^+ and y^-, we first tokenize them using the target model tokenizer. This results in fixed size embeddings y^+_{emb} and y^-_{emb}. We then compute the delta between these two embeddings. Indices for token unique to y^+_{emb}, denoted by '+' in the delta computation, are marked in m^+ as 1. Indices for token which are unique to y^-_{emb}, denoted by '-' in the delta computation, are marked in m^- as 1. Hence, the masks are computed in this manner for use in the loss function.

D Evaluation Benchmark Details

In order to assess the performance of the models in terms of security and code generation, we select six different popular evaluations benchmarks common across literature. Four of these benchmarks are for assessing the security of code generation and two of them are for assessing the performance on regular code generation of LLMs. The security benchmarks are:

Security Eval Siddiq and Santos (2022) is a dataset of code completion tasks where each prompt exposes the model to a certain CWE during code generation. It consists of 121 partially completed Python code as prompts that have been derived from examples present in various security analyzer documentation, security issue documentation or handcrafted by security experts and the authors.

Asleep at the Keyboard The authors of Pearce et al. (2022) were the first to analyse the security of code generated by LLMs. They devised 89 different code completion prompts in Python and C/C++ languages across three axes of diversity: *diversity of weakness* where the prompts are devised to expose model to certain CWEs; *diversity of prompt* where the model is exposed to a single CWE but with different variations of prompts and *diversity of domain* where the prompts are designed for hardware analysis in RTL. We select only the Python examples from diversity of weakness evaluation

set, which results in 29 prompts.

LLMSecEval (Tony et al., 2023): Tony et al. (2023) proposes the LLMSecEval dataset, which consists of 151 natural langauge instruction prompts for generating vulnerable codes in Python and C/C++. This dataset is generated by prompting various LLMs using the *Diversity of Weakness* subset of prompts from Pearce et al. (2022) and then generating natural language descriptions of the generated code using GPT-3.5. We select 81 of these instructions from the dataset which correspond to the Python programming language.

Synthetic This is a held-out test set of 200 data points from our synthesized `DiSCo`. It consists of natural language instruction prompts for generating code that are susceptible to software vulnerabilities.

For assessing code generation, we utilize the following two benchmarks:

HumanEvalX : Zheng et al. (2023) is a multilingual code generation evaluation testbed consisting of code completion prompts in multiple programming languages. It builds on top of the HumanEval (Chen et al., 2021) dataset that was originally designed for Python. Each example from the dataset consists of a docstring, function signature and public test cases that the model is prompted with and must pass. The problems are designed to assess the ability of LLMs to generate functionally correct code.

MBXP : Athiwaratkun et al. (2022) is a multilingual code generation evaluation dataset consisting of natural language instruction prompts for generating simple code and test cases for assessing them. It develops on top of the MBPP (Austin et al., 2021) dataset, which consisted of only prompts for Python programming language, by including other languages and increasing the number of testcases by 35 times. The problems are designed to be simpler than HumanEvalX and assess the fundamental programming abilities of LLMs and not any complex algorithmic programming.

E Baselines

We have the following baselinse for comparison:

SafeCoder : The authors of (He et al., 2024) developed a method of training LLMs in order to improve the security of generated code. This is done by combining instruction tuning and unlikelihood learning on a dataset of natural language intent, insecure and corresponding secure code. The model is instruction tuned to, given the intent, increase the likelihood of generating secure code and increasing the unlikelihood of generating insecure code. They also incorporate *security masking* such that only security-relevant tokens are considered during finetuning. The loss functions for this instruction tuning paradigm is as follows:

$$\mathcal{L}^{\text{sec}}(\mathbf{i}, \mathbf{o}^{\text{sec}}, \mathbf{m}^{\text{sec}}) = -\sum_{t=1}^{|\mathbf{o}^{\text{sec}}|} m_t^{\text{sec}} \odot \log P(o_t^{\text{sec}} | o_{<t}^{\text{sec}}, \mathbf{i}).$$

$$\mathcal{L}^{\text{vul}}(\mathbf{i}, \mathbf{o}^{\text{vul}}, \mathbf{m}^{\text{vul}}) = -\sum_{t=1}^{|\mathbf{o}^{\text{vul}}|} m_t^{\text{vul}} \odot \log(1 - P(o_t^{\text{vul}} | o_{<t}^{\text{vul}}, \mathbf{i})).$$

where o^{sec} and o^{vul} are secure and insecure code, i is the natural language intent, m^{sec} and m^{vul} are the security masks.

DPO : Rafailov et al. (2024) proposes the first preference optimization algorithm called Direct Preference Optimization (DPO). DPO is built on the concept of Reinforcement Learning for Human Feedback (RLHF), where a policy model is optimized using signals from a reward model. The authors of DPO show that the RLHF loss function can be reparameterized such that the policy network models the reward directly instead of requiring hosting of another reward model, thereby reducing memory requirements. The DPO loss function is as follows:

$$\mathcal{L}_{\texttt{SimPO}} = -\mathbb{E}_{(x,y^+,y^-)\sim D}[\log \sigma(\Delta)]$$
$$\Delta = \beta \log \frac{\pi_\theta(y^+|x)}{\pi_{ref}(y^+|x)} - \beta \log \frac{\pi_\theta(y^-|x)}{\pi_{ref}(y^-|x)}$$

where y^+ and y^- are the winning and losing responses, x is the prompt, π_{ref} is the unaligned base model and β is the parameter controlling deviation from base model π_{ref}.

SimPO : Meng et al. (2024) proposes a preference optimization algorithm, SimPO, that shows state-of-the-art performance on multiple benchmarks compared to other preference optimization

loss functions like DPO. The authors describe that this is due to the naturalness of the loss function to the log-likelihood function by eliminating the reference log-probabilities. Their method is also more efficient as reference modelling is not needed. The formula for the loss function is as follows:

$$\mathcal{L}_{\texttt{SimPO}} = -\mathbb{E}_{(x,y^+,y^-)\sim D}[\log \sigma(\Delta - \gamma)]$$
$$\Delta = \frac{\beta}{|y^+|}\log \pi_\theta(y^+|x) - \frac{\beta}{|y^-|}\log \pi_\theta(y^-|x)$$

where y^+ and y^- are the winning and losing responses, x is the prompt, β and γ are reward scale and target reward margin respectively.

F Setup Details

.

All the models were trained using Low-Rank Adaptation (LoRA) (Hu et al., 2022) with $r = 16$ and $\alpha = 32$. We used batch size of 32 for all the models. For Codellama, Starcoder2 and Phi-2 supervised finetuning we used larger learning rates of $1e-4$,$1e-4$ and $2e-4$ respectively while for Mistral we used $2e-5$. Supervised finetuning was done using 4-bit quantization setting for faster optimization and memory limitations. For preference optimization and LPO, we choose learning rates in the range of $[1e-5, 1e-6]$ where we observe that LPO requires a higher learning rate. For preference optimization, we use $\beta = 2.0$ and $\gamma = 0.5$ whereas for LPO, $\beta = 10.0$ and $\gamma = 5.4$ for each of the models. When we analyze LPO without the regularization, we use $\beta = 2.0$ and $\gamma = 0.5$.

For evaluation, we follow the following paradigm: for security benchmarks, we follow the generation methods of He and Vechev (2023) with a slight modification in that we sample 5 generations per evaluation sample with $T = 0.4$. For the code generation benchmarks, we sample 5 generations at $T = 0.2$ and $T = 0.6$ to calculate $pass@1$ amd $pass@5$ respectively.

All of our training and evaluation were done on 4 A6000 GPUs. Each GPU was 48 GB. It took a total of approximately 48 GPU hour in order to train and evaluate our model. For generating and refining our DiSCo dataset, it costed us approximately 100 USD on OpenAI GPT-4o API. For evaluation, it costed us around 20 USD on both OpenAI and Anthropic APIs.

G Evaluation Details

Our evaluation consist of two parts. The first part is to evaluate the security of the LLM generated code. The second part is to evaluate the code generation ability of the LLM. For each criteria, we have a different set of metrics we measure for analysis.

Evaluating Code Security Traditionally, for evaluating the security of the code generated by security, previous works incorporated the usage of the automatic security analyzers such as CodeQL(GitHub, 2024; Avgustinov et al., 2016) from GitHub in order to assess whether a piece of code is secure or insecure with respect to an certain CWE(He and Vechev, 2023; He et al., 2024). In our work, we opt for a much more comprehensive and strict evaluation of the security of code. This is done via two things. First is incorporating both CodeQL and Bandit(Bandit, 2008) security analyzers for assessing the security of the code. This makes sure that more security issues are taken into account during evaluation. This is because CodeQL and Bandit only have a subset of intersecting rules for catching security issues. Combining them both will lead to more possible patterns of security issues being identified. Second, while each example in our security testing datasets are tagged with a CWE against which we should judge whether the generation is safe or not, we opt to check for any possible security issues identified by the analyzer. This leads to more difficult but realistic evaluation of LLM generated code as you do not want to introduce new issues by overcoming previous ones and also ignore other issues that might not be relevant but exists in the generated code. Using the security analysis reports provided by CodeQL and Bandit, we first identify intersecting bugs that might lead to double counting error. We also eliminate generations for which the analyzer could parse and assess. Afterwards we calculate the following two metricsL: *Insecurity* (Insec) and *Issues per 100*.

Insecurity measures the percentage of code in the evaluation set which contain any security issues identified by our security analysis mechanism. This metric gives us a measure of the absolute reduction in insecure coding for each of the models. It is defined by the following formula:

$$Insec = \frac{insecure\ generations}{valid\ generations} \times 100$$

Our security analysis mechanism gives us a report of all the possible bugs it has identified across

the LLM generations during evaluation. We use this information to measure the mean number of issues that exist in the evaluation generations from the LLM and then multiply it by 100. This then gives us the average number of security in 100 generations or Issues per 100. This metric gives us a more nuanced understanding of the gain from each methodology analyzed in our experiments as it targets the security analysis at the bug level. The metric is defined using the following formula:

$$I@100 = 100 \times \frac{total\ bugs\ in\ generations}{valid\ generations}$$

Evaluating Code Generation To measure the utility of code generation, we use the $pass@k$ metric. Traditional $pass@k$ metric is defined as follows: you generate k code samples and if any single code passes all the testcases, you get a score of 1 or 0 otherwise. However, (Chen et al., 2021) states that calculating $pass@k$ in this manner can lead to high variance. Hence, they propose an unbiased estimator for $pass@k$, defined as follows:

$$\text{pass@}k := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

where n is the number of samples generated ($n >= k$) and c is the number of correct codes in the sample set. The idea is that out of n samples generated, a subset k will be selected for measuring $pass@k$. $n - c$ represent the number of incorrect examples in the total generation set. Hence, $\frac{\binom{n-c}{k}}{\binom{n}{k}}$ represents the probability of k containing only incorrect samples. Subtracting this value from 1 results in the calculating probability of at least 1 code in k subset that passes all the unit tests.

H CWE Analysis

Figure 4 shows the distribution of errors across the full 32 CWEs present in the analysis reports.

We can observe that CWE-78 or *OS Command Injection* is the most common CWE. CWE-259 or *hard-coded credentials* is another CWE common across all model settings. It is seen that LPO fails to fully eliminate both these error. This is likely due to their presence in DiSCo even after refinement. Many of the data points in DiSCo contain snippets of code for authenticating user credentials. As frontier LLMs generate the example code, they utilize hard-coded credentials. As a result this error passes down to the model upon training on DiSCo. For the rest of the top CWEs, LPO reduces them significantly. Whereas SafeCoder lacks cohesiveness, sometimes even increasing occurrences of certain CWEs. For CWE-89 (*SQL Injection*), we see that SafeCoder fully eliminates the problem while LPO still has them in its generations. This is likely due to the lack of enough cohesive examples for this CWE in DiSCo.

I Over-engineering via Refinement

Figure 5 shows an example where an erroneous secure code with a vulnerability is *over-engineered* for security by adding extra, unnatural layers of protection when more refinement steps take place.

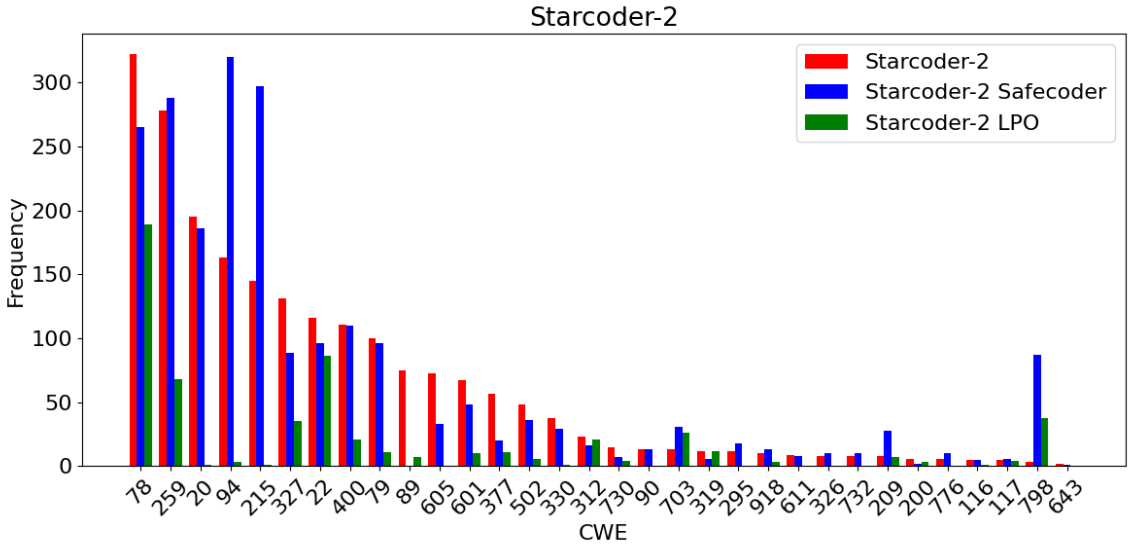

Figure 4: Full CWE Analysis for Starcoder2 baselines and LPO tuned on DiSCo for all the benchmarks.

Secure Code Snipppet After Refinement Iteration = 1

```
...

def get_file_info(filename):
    try:
        return os.popen(f"file {filename}").read()
    except Exception as e:
        return str(e)

...
```

Secure Code Snipppet After Refinement Iteration = 3

```
...
def get_file_info(filename):
    try:

        safe_filename = shlex.quote(filename)
        result = subprocess.run(['/usr/bin/file',
safe_filename], capture_output=True, text=True,
check=True)
        return result.stdout
    except subprocess.CalledProcessError as e:
        return f"Error getting file info: {e}"
    except Exception as e:
        return str(e)
...
```

Figure 5: Example of *over-engineering* due to refinement: The left portion shows a small portion of a secure code datapoint from DiSCo after one round of iteration. It contains CWE-78 vulnerability, corresponding to the *os command injection* security issue as user input is being directly executed by the os module. The right portion shows this snippet of code refined after two more rounds of refinement. We observe that the LLM has added too many extra layers of security (via lexical analysis through shlex and excess exception catches), resulting in an unnatural looking code that may deteriorate code utility when used as training data.