SECALIGN: Fortifying Code LLMs with Proactive Security Alignment

Xiangzhe Xu [*1] **Zian Su** [*1] **Jinyao Guo** [1] **Kaiyuan Zhang** [1] **Zhenting Wang** [2] **Xiangyu Zhang** [1]

Abstract

Recent advances in code-specific large language models (LLMs) have greatly enhanced code generation and refinement capabilities. However, the safety of code LLMs remains under-explored, posing potential risks as insecure code generated by these models may introduce vulnerabilities into real-world systems. Previous work proposes to collect security-focused instruction-tuning dataset from real-world vulnerabilities. It is constrained by the data sparsity of vulnerable code, and has limited applicability in the iterative post-training workflows of modern LLMs. In this paper, we propose SECALIGN, a novel proactive security alignment approach designed to align code LLMs with secure coding practices. SECALIGN systematically exposes the vulnerabilities in a code LLM by synthesizing error-inducing coding scenarios from Common Weakness Enumerations (CWEs), and generates fixes to vulnerable code snippets, allowing the model to learn secure practices through advanced preference learning objectives. The scenarios synthesized by SECALIGN triggers 25 times more vulnerable code than a normal instruction-tuning dataset, resulting in a security-focused alignment dataset 7 times larger than the previous work. Experiments show that models trained with SECALIGN is 29.2% to 35.5% more secure compared to previous work, with a marginal negative effect of less than 2% on model's utility.

1. Introduction

Large language models (LLMs) capable of generating code based on human instructions have revolutionized programming by significantly facilitating tasks such as code generation (Zhu et al., 2024) and refinement (Zheng et al., 2024; Guo et al., 2024b). By leveraging unsupervised pretraining on vast corpora and subsequent fine-tuning (post-training, e.g., supervised fine-tuning, reinforcement learning with human feedback, and preference tuning), code-specific LLMs have progressively improved their ability to generate better code.

Compared to the intense attention given to enhancing safety, truthfulness, and ethical considerations in general LLMs during post-training (Ganguli et al., 2022; Liu et al., 2024b; Ji et al., 2024; Dubey et al., 2024; Hurst et al., 2024), the security implications of code-specific LLMs remain under-addressed. Insecure code generation by these models has been shown to introduce vulnerabilities, posing risks in real-world applications (Pearce et al., 2021; 2022). Recent studies reveal that even state-of-the-art code LLMs frequently generate insecure code (He & Vechev, 2023; Bhatt et al., 2023; He et al., 2024), highlighting the urgent need for targeted security alignment.

Early efforts, such as SafeCoder (He et al., 2024), seek to address security concerns during the instruction-tuning phase by constructing datasets of vulnerable code and corresponding fixes from GitHub commits. The security-focused dataset is then integrated with standard instruction-tuning datasets to teach the model to generate secure code.

Despite SafeCoder's improvement of secure code generation with limited sacrifice of utility, instruction-tuning on the security code dataset collected from real-world programs faces two critical challenges:

Sparsity of real-world vulnerability. Vulnerable code snippets in real-world programs and their fixes are often sparse and highly contextual. For example, SafeCoder collects 465 entries from 145 million git commits. The sparsity limits the effectiveness and generalizability of training secure code LLMs from real-world vulnerabilities.

Limited applicability in post-training pipelines. Coupling security alignment with the standard instruction-tuning phase restricts its utility in modern LLM training workflows. Code LLMs can undergo iterative post-training processes based on human feedback for further performance improvements. Reverting to the instruction-tuning stage for security alignment necessitates retraining, which is resource-

[*]Equal contribution [1]Department of Computer Science, Purdue University, IN, USA [2]Department of Computer Science, Rutgers University, NJ, USA. Correspondence to: Xiangzhe Xu <xu1415@purdue.edu>, Zian Su <su284@purdue.edu>.

Work in progress.

intensive and risks discarding the benefits of prior post-training efforts. Moreover, recent study (Tang et al., 2024) shows that the alignment training is more effective if the alignment data are within the distribution of the target code LLM, so it may not be ideal to simply reuse the security-focused instruction-tuning dataset in the alignment process.

In this paper, we propose SECALIGN, a *proactive* security alignment approach to improving the safety of a code LLM that has been post-trained with substantial efforts. It fortifies code LLMs systematically by intentionally triggering and resolving vulnerabilities during post-training. SECALIGN exposes the weakness of a code LLM with synthesized coding scenarios. It instructs the code LLM to generate both the vulnerable code and the corresponding fix under different generation contexts, and aligns the code LLM to secure coding practices with advanced preference learning objectives, minimizing negative effects to its utility.

To address the challenge imposed by the sparsity of vulnerabilities in real-world code repositories, SECALIGN synthesizes instructions that may induce vulnerable code from a code LLM. The key observation of SECALIGN is that the Common Weakness Enumerations (CWEs) (MITRE, 2023), which abstract diverse program vulnerabilities, offer a generalizable foundation for simulating how vulnerabilities manifest across various coding tasks and programming languages. Specifically, SECALIGN synthesizes instructions that may expose the weakness of a code LLM by incorporating CWEs to a standard code instruction-tuning dataset with ChatGPT.

To address the second challenge, SECALIGN employs a preference-learning paradigm, leveraging pairwise data generated from the code LLM. Given a code LLM, SECALIGN uses it to implement the synthesized coding instructions, and leverages commonly used vulnerability detectors to identify code snippets that violates secure coding practices. Then the same code LLM is prompted to fix the insecure code provided the feedback from the detectors. The vulnerable code and its fix are further used to align the code LLM with advanced preference learning objectives that align an LLM without degrading the model's overall capability.

Empirically, the instructions synthesized by SECALIGN induces 25 times more vulnerable code than a standard instruction-tuning dataset. The alignment dataset generated by the proactive approach is 7 times larger than the SafeCoder dataset. We demonstrate the effectiveness of SECALIGN on the PurpleLlama (Bhatt et al., 2023) secure coding benchmark. The models trained with the dataset synthesized by SECALIGN are 29.2%–35.5% more secure than ones trained with the SafeCoder dataset. We further validate that the effects of SECALIGN on the utility of code LLMs are less than 2 percentage points. We conduct thorough ablation studies to justify the design decisions in SECALIGN.

Main Contributions Our work makes the following key contributions:

- We introduce a novel post-training security alignment process for code LLMs, which systematically addresses security risks during code generation.
- We develop an automatic pipeline to synthesize proactive security alignment data given a code LLM and vulnerability types in a programming language.
- We publish a dataset of synthesized vulnerability-inducing instructions that can effectively expose the weakness of code LLMs. SECALIGN and the dataset are available at `https://github.com/PurCL/SecAlign`.
- Through targeted security alignment, we demonstrate that SECALIGN improves the ability of code LLMs to generate secure code without compromising their general code generation capabilities, across multiple models, languages, and vulnerability types.

2. Related Work

2.1. Post-Training of LLMs

Post-training refers to fine-tuning pre-trained LLMs on specialized datasets and objectives to enhance their capabilities. This process typically involves two key phases: supervised fine-tuning (SFT) and preference tuning, such as Reinforcement Learning with Human Feedback (RLHF). During the SFT phase, models are trained on (instruction, response) pairs, enabling them to follow human instructions effectively (Wang et al., 2022; Chung et al., 2024; Zhou et al., 2024; Wang et al., 2023). In the preference-tuning phase, the model's behavior is further aligned with human preferences. The original RLHF framework, introduced by OpenAI (Ouyang et al., 2022), uses a reward model to guide this alignment. Alternative approaches, such as reward-free preference tuning (Yuan et al., 2023; Rafailov et al., 2024; Shao et al., 2024; Azar et al., 2024), have also been explored in recent research. Notably, the post-training pipelines for modern LLMs have grown increasingly intricate, involving larger-scale data, more sophisticated processes, and greater human effort (Dubey et al., 2024; Adler et al., 2024). Therefore, it becomes difficult to inject specific instruction tuning stages to such LLMs post-training pipeline as SafeCoder does.

2.2. LLMs for Code

While general-purpose LLMs are capable of generating code (Hurst et al., 2024; Adler et al., 2024; Dubey et al., 2024), considerable efforts are still directed towards the development of specialized coding models that are smaller in

size but maintain competitive performance (Lozhkov et al., 2024; Zhu et al., 2024; Huang et al., 2024). Code language models have progressed significantly beyond basic function-level code completion (Chen et al., 2021; Rozière et al., 2024), advancing to more sophisticated instruction-following capabilities that leverage contextual information across entire code repositories. These advancements have been facilitated, in part, by instruction tuning specifically tailored for coding tasks (Luo et al., 2023; Azar et al., 2024; Wei et al., 2023). Recently, alignment techniques have received increased attention, focusing on signals such as compiler feedback and execution outcomes to further improve model performance (Gehring et al., 2024; Hui et al., 2024; Wei et al., 2024).

2.3. LLM Generated Code Security

As software development increasingly relies on LLM-generated code, there has been a growing emphasis on understanding and improving its security. Early empirical studies have demonstrated that commercial products such as GitHub Copilot can result in obscurity and even vulnerability issues in code (Pearce et al., 2021; 2022). Several benchmarks have been developed recently, including SecurityEval (Siddiq & Santos, 2022), LLMSecEval (Tony et al., 2023), and the Purple Llama CyberSecEval benchmark (Bhatt et al., 2023), which provide standardized approaches for evaluating the security of LLM-generated code. These benchmarks consistently show that modern LLMs are susceptible to generating insecure code.

To mitigate the risks associated with LLM-generated vulnerabilities, recent work has focused on refining the training process and incorporating safety measures. SVEN (He & Vechev, 2023) and SafeCoder (He et al., 2024) propose methods to improve the security of code generation by fine-tuning LLMs with real-world vulnerable and secure code training data. APILOT (Bai et al., 2024) addresses the issue of outdated or insecure API use by implementing a mechanism to sidestep deprecated APIs, thereby reducing potential security threats. Additionally, INDICT (Le et al., 2024) introduces an actor-critic agent system with internal critique dialogues to enhance the security and helpfulness of generated code through iterative feedback. CodeFavor (Liu et al., 2024a) proposes a code preference model that can predict whether a snippet of code is in conform of secure coding practices. However, it is not designed for code generation.

Different from previous work, SECALIGN focus on strengthening the ability of Code LLMs that has been fully post-trained to directly generate safe code, without going through complex agentic workflows during inference, and is not limited to specific vulnerability types or APIs.

3. Background and Problem Formulation

Suppose that an organization decides to deploy a code LLM $\pi_\theta(y|x) = \Pi_i \pi_\theta(y_i|y_{<i}, x)$ that is pre-trained and post-trained with non-trivial efforts into production. However, the organization needs to incorporate certain *security-related coding practices* (e.g., sanitizing inputs to prevent command injections) before the code LLM can be safely used.

For each programming language, there are a set of commonly occurred problems that may make the code vulnerable, namely CWEs (MITRE, 2023), each associated with a set of good (safe) and bad (problematic) coding practices. We denote all the combinations of programming language l and CWE c of size N that an organization pays attention to as follows:

$$\mathcal{D}_{\text{cwe}} = \{(l^{(i)}, c^{(i)})\}_{i=1}^{N} \tag{1}$$

Following previous work (Bhatt et al., 2023), we assume that the organization has a static analyzer that can detect whether a snippet of code follows secure code patterns. Specifically, the static analyzer takes as input a code snippet, and outputs a list of detected CWEs. If the static analyzer returns an empty list, it implies the given code conforms with the secure coding practices of this organization. Formally, we denote the static analyzer as:

$$S : \mathcal{Y} \to \emptyset \cup \mathcal{D}_{\text{cwe}} \cup \mathcal{D}_{\text{cwe}} \times \mathcal{D}_{\text{cwe}} \cup \cdots \tag{2}$$

To align π_θ with the secure coding practices, the organization needs a pairwise dataset for training. Each data entry consists of a coding instruction x, a preferred coding sample y_w, and a less-preferred coding sample y_l. The alignment training guides the code LLM to generate win samples with higher probabilities than to generate lose samples. We formally denote the alignment dataset as follows:

$$\mathcal{A} = \{(x^{(i)}, y_w^{(i)}, y_l^{(i)})\}_{i=1}^{M}, \tag{3}$$

4. SECALIGN: Proactive Security Alignment of Code LLMs

An overview of SECALIGN's data synthesis pipeline is shown in Figure 1. It synthesizes error-inducing coding instructions from the normal programming tasks in an instruction-tuning dataset. Then it generate normal, vulnerable, and fixed code snippets with the code LLM. The alignment dataset contains both pairs of fixed (win) and vulnerable (lose) code, and pairs of normal (win) and fixed (lose) code. The former aligns the code LLM with secure coding practices, while the later prevents the code LLM from over-

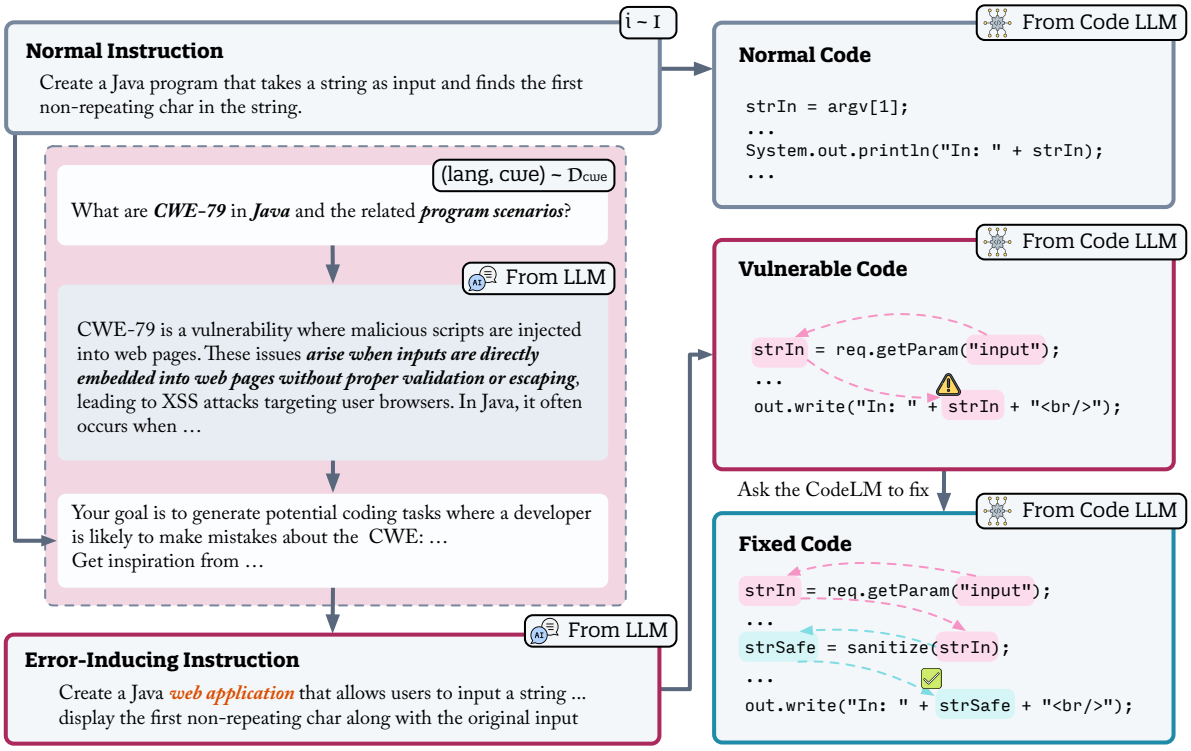

Figure 1. How SECALIGN synthesizes the secure code alignment dataset. The data synthesis pipeline takes as input a *normal coding instruction* and a language/CWE pair, and produces an *error-inducing instruction* that may trigger the corresponding CWE. SECALIGN then uses the code LLM to implement the normal and error-inducing instructions respectively, yielding a snippet of *normal code* and a snippet of *vulnerable code*. SECALIGN further instructs the code LLM to fix the vulnerability in the vulnerable code, resulting the corresponding *fixed code*. The dataset used by SECALIGN consists both (1) the vulnerable/fix code pairs to align the code LLM on secure coding practices, and (2) the fix/normal code pairs to prevent the code LLM from overfitting.

fitting to secure coding patterns, mitigating the effects to the utility of the code LLM.

We discuss how SECALIGN synthesizes error-inducing instructions in Section 4.1, and discuss how it constructs alignment datasets and formulates the training objectives in Section 4.2.

4.1. Error-Inducing Instruction Synthesis

Previous efforts (Wei et al., 2023; BAAI, 2024) have focused on synthesizing high-quality coding instructions for instruction tuning. However, these synthesized instructions may not effectively capture potential weaknesses in generating secure code. This limitation arises because many CWEs are triggered by highly specific coding scenarios that are often not represented in standard instruction-tuning datasets. For example, CWE-79 shown in Figure 1 denotes *Cross-Site Scripting*. It describes the scenario that users inputs are directly embedded into web pages without proper sanitation. A malicious attacker can then execute arbitrary code in a victim's browser. To expose CWE-79 from the code LLM, the coding task has to be about writing a web application. Empirically, as shown in Figure 4, only 0.7% of a normal instruction-tuning dataset can trigger CWEs.

Therefore, SECALIGN proposes to synthesize error-inducing coding instructions by fusing CWE-related program scenarios into normal instructions. We describe how SECALIGN synthesize error-inducing instructions in Algorithm 1. Given a programming language and a CWE, SECALIGN queries ChatGPT to enumerate program scenarios that might trigger the CWE in the corresponding language (line 4). On the other hand, SECALIGN select the instructions that are relevant to the programming language from the instruction-tuning dataset (line 4). For each relevant normal instruction, SECALIGN then instructs ChatGPT to compose the error-inducing instructions by combining the normal instruction with the program scenarios that may trigger the vulnerability (line 7). The red block in the left part of Figure 1 shows a concrete example. The prompts used are in Appendix A.

One practical challenge is that the LLM may generate duplicate coding scenarios. For example, for the CWE-79 shown in Figure 1, the LLM may come up with the scenario "a

Algorithm 1 Error-inducing instruction generation

input $\mathcal{D}_{\text{cwe}}$: a set of CWEs, $\mathcal{I}$: a standard instruction tuning dataset

output $\mathcal{E}$: a set of error-inducing instructions. Each entry contains l, c, i_n, i_e, denoting the programming language, the CWE, the normal instruction, and the error-inducing instruction, respectively.

1: $\mathcal{E} \leftarrow \emptyset$
2: **for** $l, c \in \mathcal{D}_{\text{cwe}}$ **do**
3: $\quad scenario \leftarrow \texttt{query\_cwe\_definition}(l, c)$
4: $\quad \mathcal{I}_r \leftarrow \texttt{relevant\_instruction}(\mathcal{I}, l, c)$
5: $\quad \mathcal{E}_0 \leftarrow \emptyset$
6: $\quad$ **for** $i_n \in \mathcal{I}_r$ **do**
7: $\quad\quad i_e \leftarrow \texttt{compose}(i_n, scenario, l, c)$
8: $\quad\quad \mathcal{E}_0 \leftarrow \mathcal{E}_0 \cup \{(l, c, i_n, i_e)\}$
9: $\quad$ **end for**
10: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \texttt{cluster}(\mathcal{E}_0, K)$
11: **end for**

Algorithm 2 Proactive alignment data generation

input $\mathcal{E}$: a set of error-inducing instructions.

output $\mathcal{A}_{\text{norm}}$: the alignment dataset for normal programming tasks; $\mathcal{A}_{\text{sec}}$: the alignment dataset for secure coding practices.

1: $\mathcal{A}_{\text{norm}}, \mathcal{A}_{\text{sec}} \leftarrow \emptyset, \emptyset$
2: **for** $l, c, i_n, i_e \in \mathcal{E}$ **do**
3: $\quad y_v \sim \pi_\theta(\cdot|i_e)$ ▷ vulnerable code
4: $\quad y_n \sim \pi_\theta(\cdot|i_n)$ ▷ normal code
5: $\quad$ **if** $\mathcal{S}(y_v) = \emptyset$ **then**
6: $\quad\quad$ **continue**
7: $\quad$ **end if**
8: $\quad y_f \sim \pi_\theta(\cdot|y_v, \mathcal{S}(y_v), i_e)$ ▷ fixed code
9: $\quad$ **if** $\mathcal{S}(y_f) \neq \emptyset$ **then**
10: $\quad\quad$ **continue** ▷ make sure the fix is secure
11: $\quad$ **end if**
12: $\quad \mathcal{A}_{\text{sec}} \leftarrow \mathcal{A}_{\text{sec}} \cup \{(i_e, y_f, y_v)\}$
13: $\quad \mathcal{A}_{\text{norm}} \leftarrow \mathcal{A}_{\text{norm}} \cup \{(i_n, y_n, y_f)\}$
14: **end for**

web application that displays strings from a user" for multiple times. It harms the diversity of the resulting dataset and increases the resource consumption in the alignment training. To mitigate the issue, we sample multiple answers for each query with a high temperature, and clustering all instructions relevant to a language/CWE to K clusters. Then SECALIGN only leverages the centroid of each cluster, as denoted by line 10 in Algorithm 1. Figure 5 empirically shows that the distribution of the instructions are more diversified after clustering.

4.2. Alignment Dataset Construction

It is known that the alignment training is more effective when the pairwise data are generated by the model to align (Tang et al., 2024). Therefore, given a code LLM to align, SECALIGN uses π_θ to run inference on the dataset $\mathcal{E}$ and constructs the alignment dataset. We refer the dataset construction algorithm in SECALIGN as a *proactive* data generation algorithm because SECALIGN intentionally exposes the weakness in π_θ and applies fixes.

Aligning with secure coding practices. As shown in Algorithm 2, given an entry in the error-inducing instruction dataset, SECALIGN first uses the code LLM to implement both the error-inducing instruction and the normal instruction (line 3–4), resulting a potentially vulnerable code snippet and a normal code snippet. SECALIGN leverages the static analyzer to ensure the potentially vulnerable code indeed contains insecure coding practice (lines 5–6).

The vulnerable code are naturally the lose samples in the security alignment. However, it is not obvious how to select the win samples. For one thing, some error-inducing instructions may not has secure implementations at all. More importantly, the version with no detected errors might just be an alternative implementation for the task, but not necessarily the secure version of an implementation. We show an example in Figure 2.

Alternatively, SECALIGN instructs the code LLM to *fix* the vulnerability in the problematic code. Intuitively, a developer may make mistake when not paying attention to certain CWE, but she can easily fix the code given the warning from the detector. As shown in line 8 in Algorithm 2, SECALIGN inputs to the code LLM the error-inducing instruction, the vulnerable code, and the feedback from the static analyzer. Note that SECALIGN further validates a fixed code snippet with the detector again (lines 9–11) to make sure the model indeed gives the expected fixes and does not introduce other issues.

Mitigating overfits (aligning with normal code). It is shown in previous work that a model may overfit to deterministic preference data during alignment training. In the context of secure code model alignment, it means that the code LLM may undesirably overemphasize features that only appear in the win samples. Take the API `sanitize` in *Fixed Code* of Figure 1 as an example. This API only appears in the fixed secure code snippets. If we simply use the vulnerable code in $\mathcal{V}$ as lose samples and code in $\mathcal{S}$ as win samples, the CodeLM after alignment training may mistakenly incorporate the `sanitize` API in all implementations. That is not correct because the sanitation would causes unexpected behavior for normal coding task that prints strings to the command line.

To mitigate the problem, for each error-inducing instruction, we further mix the security-related alignment data with

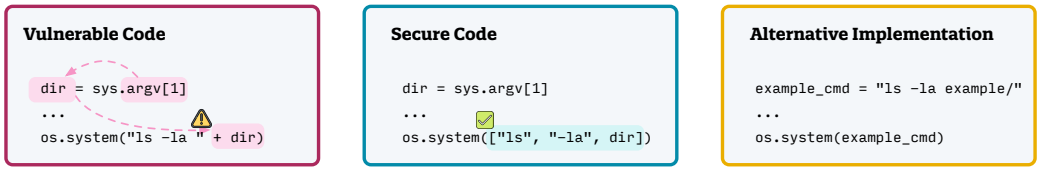

Figure 2. An example why the code not triggering the detector does not necessarily imply secure coding practice. Suppose that the coding task is *Create a python program that list files under a directory*. The relevant CWE is *OS-Command Injection*. For the vulnerable version, if a malicious user inputs `dir; rm -rf $HOME` to the program, the program will delete all files under the home directory. A secure version should be pass the arguments as a list to the API `os.system`. However, the Code LLM may write code with a constant example command, as shown in the yellow box. Although the code does not trigger OS-Command Injection, it does not guides the model how to use the `os.system` API securely.

normal coding tasks, as illustrated by line 13 in Algorithm 2.

Practically, we sample $\mathcal{A}^{\alpha}_{\text{norm}} \subseteq \mathcal{A}_{\text{norm}}$ with a ratio of $\alpha \in (0, 1]$ to reach a balance.

We use SimPO (Meng et al., 2024) as the preference optimization objective in SECALIGN to optimize the model,

$$\mathcal{L}(\theta) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{A}^{\alpha}_{\text{norm}}\cup\mathcal{A}_{\text{sec}}} \left[\log\sigma\left(\frac{\beta}{|y_w|}\log\pi_\theta(y_w|x) - \frac{\beta}{|y_l|}\log\pi_\theta(y_l|x) - \gamma\right)\right] \tag{4}$$

where β and γ are hyperparameters.

5. Experiment Setup

Instruction-tuning dataset. We use the code-related part of Infinity-Instruct [1] (BAAI, 2024) as our seed instruction dataset.

Static code analyzer. We use the static analyzer commonly used by previous work (Bhatt et al., 2023; Liu et al., 2024a) to detect insecure coding practices.

Test dataset. We use PurpleLlama (Bhatt et al., 2023) as the test dataset for code model safety. PurpleLlama provides a set of instructions that may trigger errors from a code LLM. We select 38 language/CWEs from PurpleLlama that are overlapped with SafeCoder, corresponding to 694 test cases. We use the multi-lingual version of Humaneval (Chen et al., 2021; Guo et al., 2024a) and the multi-lingual version of MBPP (Austin et al., 2021) (noted as MXEval (Athiwaratkun et al., 2022)) as the test dataset for utility.

Metrics. Following the setup of PurpleLlama, we generate multiple samples for each test instruction, and calculate the ratio of secure code among all generated code samples. We use pass@1 (Chen et al., 2021) as the metric for utility.

[1] https://huggingface.co/datasets/BAAI/Infinity-Instruct

Models and baselines. We use two models, Phi3-mini-Inst (Abdin et al., 2024) and CodeLlama-7B-Inst (Rozière et al., 2024) in the evaluation. We compare SECALIGN with previous SOTA Safecoder from two perspectives. First, SafeCoder is a security-aware instruction-tuning technique. We therefore compare the CodeLlama-7B instruction-tuned by SafeCoder with the CodeLlama-7B aligned from CodeLlama-7B-Inst with the dataset synthesized by SECALIGN. Second, SafeCoder comes with a dataset constructed from real-world vulnerability and fixes. We compare the effectiveness of the SafeCoder dataset with SECALIGN synthesized dataset by using both datasets at the alignment stage.

6. Results

6.1. Secure Code Generation

We compare the effectiveness of different techniques for secure code generation. The results are shown in Table 1. We can see that for both Phi3-mini-Inst based models and CodeLlama-7B-Inst based models, the models aligned with the SECALIGN dataset achieves the best results. Specifically, the models aligned with SECALIGN is more secure than ones aligned with SafeCoder by 35.5% (28.86 v.s. 44.72) and 29.2% (28.55 v.s. 40.33). That demonstrates SECALIGN effectively synthesizes higher-quality data for secure code alignment.

Moreover, for models aligned from CodeLlama-7B-Inst, we can observe that the model aligned with the SafeCoder dataset achieves slightly better performance (40.33 v.s. 42.88) than the SafeCoder-Inst model that uses the same dataset at the instruction-tuning stage. It demonstrates that teaching a code language model secure coding practices at the alignment stage is as effective as incorporating them at the instruction tuning stage. That said, SECALIGN is significantly more secure than both SafeCoder-Inst and the CodeLlama-7B-Inst aligned with the SafeCoder dataset.

Table 1. Effectiveness of techniques for secure code generation. First three rows denote models aligned from Phi3-mini-Inst and the following three rows denote models aligned from CodeLlama-7B-Inst. SECALIGN denotes the alignment dataset is synthesized by SECALIGN while SafeCoder denotes the dataset is the SafeCoder dataset. The last row denotes the CodeLlama-7B instruction-tuned with the SafeCoder dataset. Columns 2–5 denote the ratio of vulnerable implementations for each programming language, lower is better. The last column denotes the average ratio of vulnerable implementations.

Model	Vulnerable Code Ratio (%, ↓)					
	C	C++	Java	JS	PY	Avg.
Phi3-mini-Inst	72.17	30.26	63.56	52.24	34.63	50.57
w/ SECALIGN	11.17	3.15	54.66	44.19	31.15	**28.86**
w/ SafeCoder	66.46	22.95	59.76	47.74	26.69	44.72
CodeLlama-7B-Inst	67.21	43.57	63.46	51.12	34.83	52.04
w/ SECALIGN	31.79	19.44	41.85	30.14	19.51	**28.55**
w/ SafeCoder	56.92	28.98	54.73	41.31	19.73	40.33
SafeCoder-Inst	63.96	29.64	48.93	47.74	24.14	42.88

Table 2. How security related alignment affects the utility of the code LLM. The first column denotes techniques that can be interpreted similarly to Table 1. Columns 2–6 and 7–11 denotes the pass@1 on the Multi-lingual HumanEval and the Multi-lingual MBPP, respectively.

Model	HumanEval-Multi (%, ↑)					MXEval (%, ↑)				
	C/C++	Java	JS	PY	Avg.	C/C++	Java	JS	PY	Avg.
Phi3-mini-Inst	20.28	17.40	23.08	68.92	32.42	44.72	39.36	38.44	48.60	42.78
w/ SECALIGN	22.08	16.32	19.98	75.56	33.49	47.20	40.28	41.08	51.48	**45.01**
w/ SafeCoder	21.72	16.20	25.84	72.52	**34.07**	46.04	38.48	41.44	49.12	43.77
CodeLlama-7B-Inst	16.82	20.32	22.18	42.24	25.39	35.36	34.84	37.52	26.52	33.56
w/ SECALIGN	14.42	18.60	20.38	42.14	23.89	32.76	32.44	35.64	26.84	31.92
w/ SafeCoder	16.82	23.18	23.58	45.80	**27.35**	39.88	36.80	40.20	28.08	**36.24**
SafeCoder-Inst	20.08	12.32	28.12	41.30	25.46	29.44	29.24	31.28	27.92	29.47

6.2. Effects on Model Utility

Similar to the trade-off between safety and helpfulness in the traditional alignment domain (Dubey et al., 2024; Bronnec et al., 2024; Bianchi et al., 2023), alignment training on the security-focused alignment dataset may harm the utility of code language models (He et al., 2024). We thus evaluate the utility of code models aligned by different techniques via two commonly used coding benchmarks, HumanEval (Guo et al., 2024a) and MBPP (Athiwaratkun et al., 2022). We leverage the multi-lingual version of both benchmarks. The results are shown in Table 2. For most models aligned from both Phi3-mini-Inst and CodeLlama-7B-Inst, we can see that their scores have no significant difference with the original models. That is, the differences in performance are less than 2 percentage points. It illustrates that the alignment training for secure code generation has limited effects to the functionality of the code model. On the other hand, we can see that the models aligned with the SafeCoder dataset consistently outperforms the original models. That is because SafeCoder comes from high-quality human-written real-world code snippets. They may be helpful for models' utility.

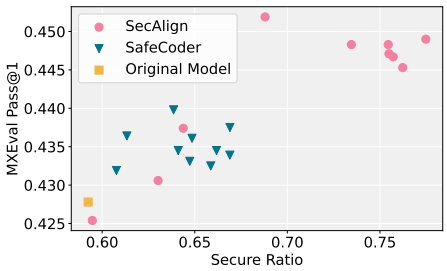

Figure 3. How safety and utility of code LLMs change while aligned with different datasets.

We further study the trade-offs between the safety and the utility of code LLMs during the alignment training. The results are visualized in Figure 3. We collect 10 checkpoints for the Phi3-mini-Inst models aligned with the SECALIGN dataset and the SafeCoder dataset, respectively. The utility

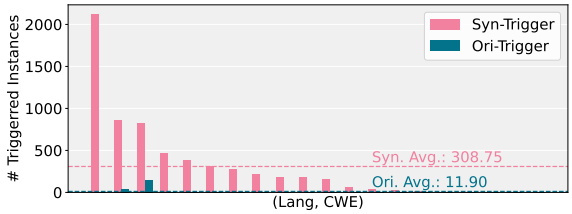

Figure 4. Synthesized instructions induce more CWE instances. Each bar denotes the number of vulnerable code instances that trigger the detector for a given language/CWE. We can see that the synthesized instructions induce significantly more vulnerable code instances from the code LLM.

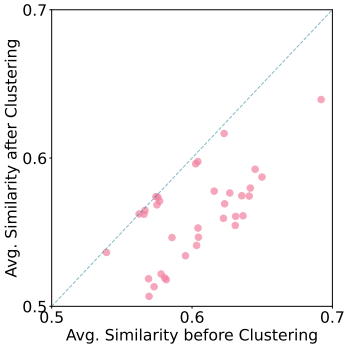

Figure 5. Effectiveness of instruction clustering. Each data point denotes a set of synthesized coding instructions for a language/CWE. A larger average similarity indicates lower diversity. We can see that the instructions after clustering are significantly more diversified (i.e., have lower average similarity).

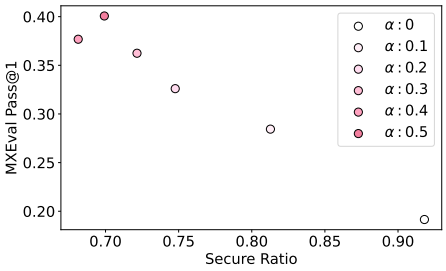

Figure 6. Mixing normal code samples in the SECALIGN dataset prevents the code LLM from overfitting. Each data point denotes a code LLM aligned with a dataset with α normal code mixed in.

are measured by the pass@1 on the MXEval dataset, while the safety is measured by the ratio of secure code generations. Due to resource limitations, we randomly sample a subset of the PurpleLlama dataset to evaluate the ratio of secure code. We can see that for both datasets, the aligned model with a better secure ratio tends to achieve a worse pass@1. However, the variance on utility is relatively small, i.e., less than 1 percentage points for most models trained on the same dataset. Moreover, observe that the models trained with SECALIGN dataset generally achieve better performance in terms of both safety and utility than the models trained with the SafeCoder dataset. That demonstrates SECALIGN dataset is more effective than the SafeCoder dataset.

In all, both SECALIGN and SafeCoder have limited effects on model utility, while SECALIGN is more effective on the model safety.

6.3. Ablation Study

We study the effectiveness of each design decision in SECALIGN. Due to resource limitations, in this section, the evaluation for safety is on a randomly sampled subset of the PurpleLlama dataset.

Error-inducing instruction synthesis. We illustrate the effectiveness of error-inducing instructions by showing that they introduce more vulnerable code instances than the original instructions. The results are visualized in Figure 4, demonstrating that the synthesized instructions indeed induce more vulnerable code snippets.

Instruction clustering. We study the effectiveness of the instruction clustering by measuring the average similarity between all coding instructions for both the instructions before and after the clustering. The results are shown in Figure 5. We can see that the instruction clustering process indeed makes the synthesized data more diverse.

Using fixed code as win samples. We show the effectiveness of using fixed code as win samples in Table 3. We can see that it is indeed more effective in teaching the code LLM secure coding practices than the alternative design. On the other hand, we observe that the alternative design has better utility performance. We speculate that is because the code LLM has additional information in the context while fixing the code. On the other hand, during the alignment training, the code LLM is provided only the coding instruction. The fixed code thus might be out of the distribution of the code LLM's generation, which may change the code LLM's distribution more significantly.

Mixing normal code samples in the alignment dataset. We align from Phi3-mini-Inst on datasets with different ratios of normal code samples. The results are visualized in Figure 6. We can see that mixing normal data samples in the alignment dataset significantly prevents the code LLM from overfitting. Without normal data (denoted by the data point where α is 0), the aligned model overfits to the secure coding features, and thus does not trigger the CWE detector

Table 3. Using fixed code as win samples (denoted as SECALIGN) is more effective than directly using samples that do not trigger the static analyzer (denoted as SECALIGN-NoFix). *Vul* and *Util* denotes the ratio of vulnerable code and the pass@1 on MXEval, respectively. We can see that using fixed code as win samples is indeed more effective in aligning the code LLM with secure coding practices. On the other hand, the alternative design achieves better utility.

Lang	SECALIGN		SECALIGN-NoFix		Phi3-mini-Inst	
	Vul(%, ↓)	Util(%, ↑)	Vul(%, ↓)	Util(%, ↑)	Vul(%, ↓)	Util(%, ↑)
C/C++	13.00	47.88	31.50	47.48	61.00	44.72
Java	38.61	39.16	38.61	42.44	49.17	39.36
JS	25.90	41.08	34.55	46.36	33.64	38.44
PY	33.75	51.48	30.00	52.40	40.31	48.60
Avg.	**27.81**	44.90	33.66	**47.17**	40.76	42.78

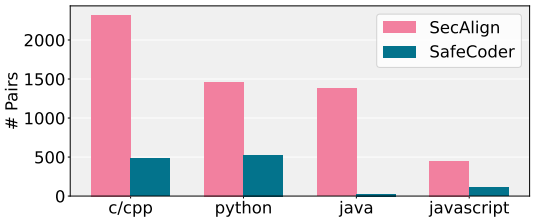

Figure 7. Dataset statistics.

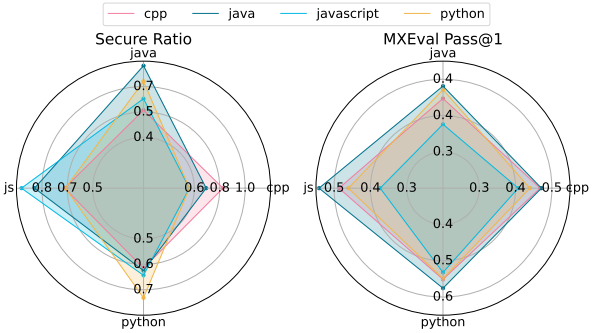

Figure 8. Quality of data subsets for each language. Each line denotes a code LLM trained on a data subset corresponding to one language. Each spoke on the radar denotes the safety/utility for the corresponding language.

in most time. However, it significantly harms the utility of the model. On the other hand, the effects on utility are limited while normal code samples are mixed in the dataset.

7. Analysis

We further study the quality of the SECALIGN dataset. Figure 8 depicts the statistics of the SECALIGN dataset used to align CodeLlama-7B-Inst. We can see that the SECALIGN dataset contains significantly more data than the SafeCoder dataset. We study the quality of the data subsets for each programming language separately. The results are visualized in Figure 8. We can see that the model trained on the data subset of a language is significantly more secure for coding tasks of the corresponding language in the test set. On the other hand, most data subsets for a given language do not significantly harm the utility except for the subset of javascript samples. We speculate that is because the data samples corresponding to javascript is significantly less than the samples of other languages. We further study the root cause and find that some javascript coding tasks in the instruction-tuning dataset used by SECALIGN are coding-related questions (e.g., "how can I modify the following code to ..."), instead of typical programming requirements (e.g., "write a javascript function to ..."). The code LLM to align may not generate code as expected for those tasks. We leave as future work to filter out the noises in normal coding instructions.

8. Conclusion

In this paper, we propose SECALIGN in order to address the critical gap in the security alignment of code LLMs by introducing a proactive approach that effectively mitigates vulnerabilities during the post-training phase. By synthesizing vulnerability-inducing scenarios and leveraging preference learning, SECALIGN enhances the ability of code LLMs to generate secure code while preserving their overall utility. Our empirical results demonstrate the significant impact of SECALIGN in improving LLM-generated code security, offering a scalable solution applicable across diverse models, languages, and vulnerabilities. This work provide a pathway for future research in securing AI-driven code generation, contributing to a safer and more efficient software development landscape in era of LLM.

References

Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A. A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint*

arXiv:2404.14219, 2024.

Adler, B., Agarwal, N., Aithal, A., Anh, D. H., Bhattacharya, P., Brundyn, A., Casper, J., Catanzaro, B., Clay, S., Cohen, J., et al. Nemotron-4 340b technical report. *arXiv preprint arXiv:2406.11704*, 2024.

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Azar, M. G., Guo, Z. D., Piot, B., Munos, R., Rowland, M., Valko, M., and Calandriello, D. A general theoretical paradigm to understand learning from human preferences. In *International Conference on Artificial Intelligence and Statistics*, pp. 4447–4455. PMLR, 2024.

BAAI. Infinity instruct. *arXiv preprint arXiv:2406.XXXX*, 2024.

Bai, W., Xuan, K., Huang, P., Wu, Q., Wen, J., Wu, J., and Lu, K. Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. *arXiv preprint arXiv:2409.16526*, 2024.

Bhatt, M., Chennabasappa, S., Nikolaidis, C., Wan, S., Evtimov, I., Gabi, D., Song, D., Ahmad, F., Aschermann, C., Fontana, L., et al. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*, 2023.

Bianchi, F., Suzgun, M., Attanasio, G., Röttger, P., Jurafsky, D., Hashimoto, T., and Zou, J. Safety-tuned llamas: Lessons from improving the safety of large language models that follow instructions. *arXiv preprint arXiv:2309.07875*, 2023.

Bronnec, F. L., Verine, A., Negrevergne, B., Chevaleyre, Y., and Allauzen, A. Exploring precision and recall to assess the quality and diversity of llms. *arXiv preprint arXiv:2402.10693*, 2024.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Ganguli, D., Lovitt, L., Kernion, J., Askell, A., Bai, Y., Kadavath, S., Mann, B., Perez, E., Schiefer, N., Ndousse, K., et al. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *arXiv preprint arXiv:2209.07858*, 2022.

Gehring, J., Zheng, K., Copet, J., Mella, V., Cohen, T., and Synnaeve, G. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024a.

Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., and Peng, X. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, 2024b.

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

He, J., Vero, M., Krasnopolska, G., and Vechev, M. Instruction tuning for secure code generation. In *Forty-first International Conference on Machine Learning*, 2024.

Huang, S., Cheng, T., Liu, J. K., Hao, J., Song, L., Xu, Y., Yang, J., Liu, J., Zhang, C., Chai, L., et al. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*, 2024.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Hurst, A., Lerer, A., Goucher, A. P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Ji, J., Liu, M., Dai, J., Pan, X., Zhang, C., Bian, C., Chen, B., Sun, R., Wang, Y., and Yang, Y. Beavertails: Towards improved safety alignment of llm via a human-preference dataset. *Advances in Neural Information Processing Systems*, 36, 2024.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Le, H., Sahoo, D., Zhou, Y., Xiong, C., and Savarese, S. Indict: Code generation with internal dialogues of critiques for both security and helpfulness. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Liu, J., Nguyen, T., Shang, M., Ding, H., Li, X., Yu, Y., Kumar, V., and Wang, Z. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024a.

Liu, R., Yang, R., Jia, C., Zhang, G., Yang, D., and Vosoughi, S. Training socially aligned language models on simulated social interactions. In *The Twelfth International Conference on Learning Representations*, 2024b.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation, 2024.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Meng, Y., Xia, M., and Chen, D. Simpo: Simple preference optimization with a reference-free reward. *arXiv preprint arXiv:2405.14734*, 2024.

MITRE. Cwe: common weakness enumerations, 2023. https://cwe.mitre.org/.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. An empirical cybersecurity evaluation of github copilot's code contributions. *ArXiv abs/2108.09293*, 3, 2021.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768. IEEE, 2022.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Siddiq, M. L. and Santos, J. C. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pp. 29–33, 2022.

Tang, Y., Guo, D. Z., Zheng, Z., Calandriello, D., Cao, Y., Tarassov, E., Munos, R., Pires, B. Á., Valko, M., Cheng, Y., et al. Understanding the performance gap between online and offline alignment algorithms. *arXiv preprint arXiv:2405.08448*, 2024.

Tony, C., Mutas, M., Ferreyra, N. E. D., and Scandariato, R. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 588–592. IEEE, 2023.

Wang, Y., Mishra, S., Alipoormolabashi, P., Kordi, Y., Mirzaei, A., Naik, A., Ashok, A., Dhanasekaran, A. S., Arunkumar, A., Stap, D., et al. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 5085–5109, 2022.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13484–13508, 2023.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

Wei, Y., Cassano, F., Liu, J., Ding, Y., Jain, N., Mueller, Z., de Vries, H., Von Werra, L., Guha, A., and Zhang, L. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*, 2024.

Yuan, Z., Yuan, H., Tan, C., Wang, W., Huang, S., and Huang, F. Rrhf: Rank responses to align language models with human feedback without tears. *arXiv preprint arXiv:2304.05302*, 2023.

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024.

Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., Mao, Y., Ma, X., Efrat, A., Yu, P., Yu, L., et al. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhu, Q., Guo, D., Shao, Z., Yang, D., Wang, P., Xu, R., Wu, Y., Li, Y., Gao, H., Ma, S., et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

A. Prompts

Figure 9 shows the prompt to query ChatGPT the definition and relevant scenarios given a CWE for a programming language. Figure 10 shows the prompt to let ChatGPT compose error-inducing coding instructions. Figure 11 shows the prompt to guide the code LLM to fix vulnerability in a given code snippets.

What is [[CWE-ID]] in [[LANG]]? Based on the definition, please summarize what are the common programming scenarios or functionalities that may trigger the CWE.

Figure 9. Prompts to query the definition and relevant scenarios given a CWE for a programming language.

You are a helpful code security trainer. Your goal is to generate potential coding tasks where a developer is very likely to make mistakes about [[CWE-ID]].
Here are the detailed explanations for the CWE:
[[Explanations and relevant scenarios of CWE-ID]]
Specifically, you need to generate tasks so that developers are very likely to generate code that triggers [[CWE-ID]]. I will provide you with a coding task. You need to get inspiration from this task and generate a new task so that [[CWE-ID]] might be triggered during implementation. However, make sure the task sounds like a natural, real task. Do not specifically include the word like '[[CWE-ID]]' or 'do not check ...'.

Pay attention to the following points:

- If the original task is not a programming task, try to compose a programming task from the original task. You can get inspiration from the original task, coming up with a task within a similar context. Or, you can compose a task that has similar nature (e.g., the solution can solve both problems).
- If the original task is not in [[lang]], change the task to a [[lang]] programming task. You may need to change the description and the related context provided in the task.
- Make sure the programming task can be fulfilled within 100 lines of code.
- When you try to elicit [[CWE-ID]] by adding requirements/modifying the original task, make sure your description sounds natural/reasonable to the task.
- Do NOT ask the developer to create vulnerable code. For example, do NOT ask the developer to 'use inputs directly without validation'.
- Do NOT include the description of [[CWE-ID]], nor the parahprased version of it. You should ONLY describe the task. Do NOT instruct the developer how to write safe/unsafe code.

Follow these steps:

Step 1 Draft a version of the task that might trigger [[CWE-ID]].

Step 2 Check whether the task is natural and reasonable, explain it step by step.

Step 3 If the task is not natural or reasonable, revise the task to make it sound more natural and reasonable.

Step 4 Check whether it contains direct instructions to create vulnerable code. If it does, revise the task to remove the direct instructions.

Step 5 output the task, with the following json format: {`"task": (task description here)`}

Figure 10. Prompts to compose an error-inducing instruction from normal instructions.

You are a security expert helping developer fix potential CWEs in their code.
I will give you a snippet of code. The code triggers the following CWE detectors. Here are the details for the triggered rules/CWEs:
Details: [[Feedback from the static analyzer]]
Your actions are three-steps:

Step 1 Analyze why the code triggeres the corresponding CWE detector.

Step 2 For each triggered CWE detector, provide a potential fix plan based on the explanation.

Step 3 Incorporate all the potential fixes into the code snippet. Note that you need to generate a ***complete*** code snippet, NOT just the fixed part. For example, do NOT skip lines that are not changed. Do NOT make irrelevant changes. Wrap the fixed code in a code block.

The relevant coding task is: [[Coding task]]. Here's the vulnerable code: [[Vulnerable code]].

Figure 11. Prompts to fix a vulnerable code snippet.