# Secure-Instruct: An Automated Pipeline for Synthesizing Instruction-Tuning Datasets Using LLMs for Secure Code Generation

**Junjie Li**[1], **Fazle Rabbi**[1], **Bo Yang**[1], **Song Wang,**[2] **Jinqiu Yang**[1]

[1]Concordia University
[2]York University

{l_unjie, b_yang20}@encs.concordia.ca, fazle.rabbi@mail.concordia.ca, wangsong@yorku.ca, jinqiu.yang@concordia.ca

**Abstract**

Although Large Language Models (LLMs) show promising solutions to automated code generation, they often produce insecure code that threatens software security. Current approaches (e.g., SafeCoder) to improve secure code generation are limited by small, imbalanced instruction-tuning datasets. In this work, we present Secure-Instruct, a novel pipeline that automatically synthesizes high-quality vulnerable and secure code examples and instruction-tunes LLMs to align task description and secure code generation abilities.

We evaluate Secure-Instruct on four representative LLMs using two security-related benchmarks: our own CWEBench and the existing CWEval. CWEBench comprises 93 scenarios on 44 CWEs, all without overlap with Secure-Instruct's synthetic instruction-tuning dataset, while CWEval covers 31 CWEs with 119 manually verified security-critical tasks. We find that Secure-Instruct improves both security and functional correctness in code generation. On CWEBench, Secure-Instruct substantially improves secure code generation, giving a 28.5% increase on average in secure ratio over the pre-trained models and outperforms SafeCoder by 12.6%. On CWEval, Secure-Instruct achieves an increase of 157.3% for CodeLlama-7B and 46.4% for Mistral-7B in Func-Sec@1 over pretrained models, and significantly outperforms SafeCoder.

## 1 Introduction

Large Language Models (LLMs), trained on billions of parameters across extensive corpora, have significantly advanced various software engineering tasks, such as code generation (Wei et al. 2024; Luo et al. 2023; Lin et al. 2024) and program repair (Jin et al. 2023; Joshi et al. 2023). Prominent models such as ChatGPT (OpenAI 2024), Llama (Touvron et al. 2023), and GitHub Copilot (GitHub 2024) have recently gained widespread popularity for their effectiveness in these domains.

Despite their success, LLMs often generate code with security vulnerabilities. Pearce et al. (Pearce et al. 2022) revealed that approximately 40% of code snippets generated by GitHub Copilot contain vulnerabilities, and similar issues have been reported in other models (Liu et al. 2024; Sandoval et al. 2023). These findings suggest that, despite advances, vulnerabilities continue to pose a significant challenge in LLM-based code generation.

Several efforts (Hajipour et al. 2023; Wang et al. 2023; He and Vechev 2023; He et al. 2024; Li et al. 2024a,b) have been made to enhance the secure code generation capabilities of LLMs. Among all, SafeCoder (He et al. 2024) is the state-of-the-art (SOTA), which leverages real vulnerability fixes for instruction-tuning LLMs and outperforms other approaches. Yet, techniques like SafeCoder face two major limitations. First, exploiting real-world vulnerabilities and their fixes is limited by scale due to their rarity. For instance, SafeCoder managed to collect only 465 fixing examples covering 23 different Common Weakness Enumerations (CWEs) with 6 programming languages from *145 million commits*. This not only highlights the inefficiency of the approach but also its limited scalability, as even an extensive dataset contains examples for only a small subset of CWEs. Second, the methodology relies on a non-trivial manual inspection in the post-processing phase to balance and clean the dataset. Specifically, it down-samples over-represented CWE cases and manually removes examples that were incorrectly collected by the automated pipeline. Consequently, only 465 high-quality examples were retained from the 1,211 initially collected instances.

To address the abovementioned challenges, we propose Secure-Instruct, **a novel and automated pipeline designed to synthesize instruction tuning datasets from descriptive documentation of CWEs**. Specifically, we utilize the CWE descriptions and code examples provided by the MITRE (MITRE 2025) database as a foundational resource to prompt LLMs to generate diverse, high-quality, and secure code examples that LLMs can easily understand for later instruction-tuning. Secure-Instruct integrates static security detection tools (SonarQube and CodeQL) to automatically ensure the high quality of LLM-synthesized datasets. In addition, inspired by recent works in instruction tuning (Longpre et al. 2023), our proposed Secure-Instruct leverages instruction tuning to better align LLMs with secure code generation. Secure-Instruct (1) bridges the gap between CWE documentation and practical secure code generation, (2) improves the performance of secure code generation in LLMs, (3) requires no manual efforts in curating high-quality instruction-tuning datasets, and (4) can be easily extensible to work on new CWEs as it does not

rely on open-source repositories for curating high-quality instruction-tuning dataset for new CWEs.

Secure-Instruct experiments with two synthetic data generation schemes: (1) vulnerable-code driven and (2) secure-code driven. For (1), given a CWE description document, Secure-Instruct prompts an LLM to generate vulnerable code first, then continuously fixes the vulnerable code to achieve secure code, resulting in code pairs of *(vul, secure)*. For (2), Secure-Instruct prompts an LLM to generate secure code practices directly *(secure)*. Then, Secure-Instruct leverages LLMs to generate an instruction for each code snippet. Last, Secure-Instruct employs two instruction tuning strategies: one is the standard one (Wei et al. 2024), and the other is the secure instruction-tuning approach proposed by He et al. (He et al. 2024). This approach uses an instruction-tuning technique designed to learn secure coding patterns and assign penalties to vulnerable code snippets. We evaluate Secure-Instruct on four representative LLMs (CodeLlama-7B, CodeGen2-7B, Mistral-7B, and StarCoder-1B), which are commonly used in prior works on LLM secure code generation (He and Vechev 2023; He et al. 2024) for fair comparison and are suitable under limited computing resources. Our evaluation includes two benchmarks, *CWEBench* (94 security scenarios covering 44 CWEs), and *CWEval* (Peng et al. 2025) (119 manually verified security-critical tasks covering 31 CWEs). On *CWEBench*, Secure-Instruct outperforms the state-of-the-art secure code generation approach, SafeCoder (He et al. 2024) by a large margin, with an average improvement of 12.6%. On *CWEval*, Secure-Instruct achieves better performance with an average 56.6% improvement of Func-Sec@1 (both secure and functionally correct) compared to pre-trained models, and it outperforms SafeCoder with an average improvement of 140.8%. Finally, it also improves functional correctness with +39.6% (CodeLlama-7B), +12.6% (Mistral-7B), and +6.5% (CodeGen2-7B) on HumanEval (Chen, Tworek et al. 2021). These results highlight the effectiveness of Secure-Instruct in advancing both secure and functional code generation.

**Contributions. 1)** We propose Secure-Instruct, a fully automated framework that synthesizes high-quality datasets from CWE documentation to instruction tune LLMs for secure code generation. Secure-Instruct is generalizable and scalable to a diverse set of CWEs and is completely free of manual effort. **2)** We extend existing secure benchmarks to a total of 93 security-sensitive prompts covering 44 CWEs to provide a comprehensive evaluation on secure code generation. **3)** We conduct a comprehensive evaluation for Secure-Instruct on four LLMs, and we take a comparative analysis of secure performance with the SOTA approach. Our approach not only significantly improves the secure code generation but also improves the functionality correctness evaluated on CWEval, HumanEval, and MBPP benchmarks. **4)** We share the synthesized instruction-tuning dataset, the replication code, and models' weights at https://zenodo.org/records/16715355.

```
1 int main(int argc, char *argv[]) {
2     // Get the username from the command
          line
3     char *username = argv[1];
4     // Get the user info from the system
5     char command[100] = {0};
6     // Execute the command using the
          username as input
7     // ** The code generated below **
8     sprintf(command, "id %s", username);
9 }
```

Figure 1: An example of vulnerable code generation. Line 8 is generated by CodeLlama-7B.

## 2 Background and Related Work

**Security Concerns of LLM-Generated Code.** While LLMs show promise in automating code generation, their training on unsanitized open-source data often leads to insecure code. Figure 1 shows an example of vulnerable code generated by CodeLlama-7B. This vulnerable code (line 8) demonstrates a CWE-78 ('OS Command Injection'). 'username' comes directly from 'argv[1]', which is not sanitized or validated. An attacker could inject arbitrary commands by passing a specially crafted username as input.

Several studies (Sandoval et al. 2023; Siddiq and Santos 2023; Gong et al. 2024; Tihanyi et al. 2025; Fu et al. 2023; Mohsin et al. 2024) have investigated the security vulnerabilities present in LLM-generated code. Pearce et al. (Pearce et al. 2022) evaluated GitHub Copilot, which is built on the OpenAI Codex model, and reported that approximately 40% of the generated code contained security vulnerabilities. These vulnerabilities are systematically classified under the Common Weakness Enumeration (CWE) (CWE-MITRE 2022), a widely adopted category system for security vulnerabilities. Khoury et al. (Khoury et al. 2023) examined ChatGPT in security-critical coding scenarios, discovering that the model produced insecure code in 16 cases, with only 7 cases corrected after iterative interaction with the model. Similarly, Perry et al. (Perry et al. 2023) observed that developers relying on AI-assisted tools are more likely to introduce security vulnerabilities.

**Improving the Security of LLM Generated Code.** To overcome these limitations, Li et al. (Li et al. 2024b) proposed an approach to fine-tune one LLM by using the vulnerability-fixing commits. Hajipour et al. (Hajipour et al. 2024) introduced a technique to synthesize secure code examples guided by CodeQL (GitHub 2023) oracle. Another work (Li et al. 2024a) explored the effect of parameter-efficient fine-tuning techniques in secure code generation. He et al. (He and Vechev 2023) introduced SVEN, an advanced prompting method, i.e., prefix-tuning, that uses property-specific continuous vectors (prefixes) to guide LLMs to generate secure code and reduce the likelihood of generating vulnerable code. The training of SVEN optimizes these continuous vectors using a carefully curated, high-quality dataset.

Most recently, SafeCoder (He et al. 2024) applies instruction-tuning on a dataset of 465 vulnerable-fix pairs,

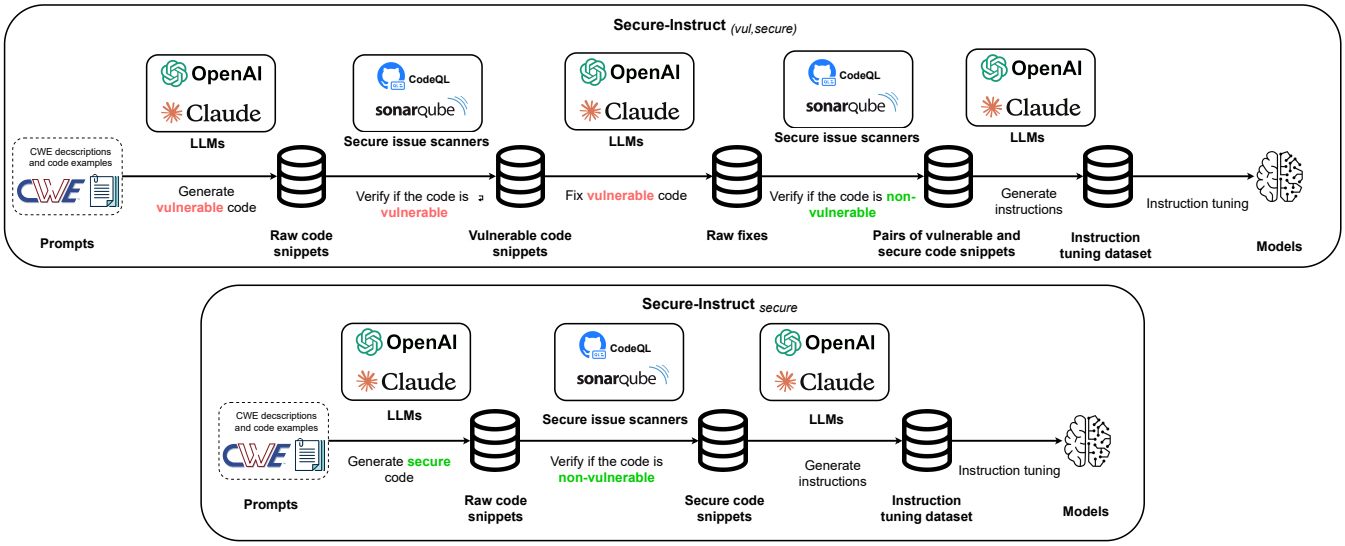


Figure 2: An overview of Secure-Instruct.

achieving state-of-the-art results in secure code generation. Despite improved results, SafeCoder has several limitations. First, the quality of code from open-source repositories is inconsistent, often involving complex dependencies with external libraries. As a result, many code snippets are unsuitable for fine-tuning, requiring significant manual effort to filter and curate appropriate samples. Second, it is challenging to obtain a sufficiently large fine-tuning dataset for each category of security issues (e.g., CWE). For example, the work (He et al. 2024) identified 71 code snippets for CWE-079 but only one for CWE-732, resulting in an imbalance that hinders the model's ability to achieve robust security performance. Besides, this approach is expensive. They only collected 456 code snippets covering 23 CWEs from 145 million commits, making it difficult to expand to other CWEs. To address these limitations, we propose Secure-Instruct to automatically construct a more extensive and diverse dataset to further enhance the security of LLM-generated code.

## 3 The Methodology of Secure-Instruct

We introduce Secure-Instruct that leverages LLMs (e.g., ChatGPT and Claude) to synthesize data for instruction-tuning code language models to enhance secure code generation. Figure 2 shows an overview of Secure-Instruct. It adopts two data-generation schemes, secure-code driven (Secure-Instruct$_{secure}$) and vulnerable-code driven (Secure-Instruct$_{(vul,\ secure)}$), to generate high-quality secure training data with no human intervention. It then uses LLMs to generate natural-language instructions that describe the functionality of each code example in the dataset. Finally, Secure-Instruct applies two instruction-tuning strategies to produce security-enhanced LLMs (Section 3.3): standard instruction tuning (Longpre et al. 2023) and security-specialized instruction tuning (He et al. 2024).

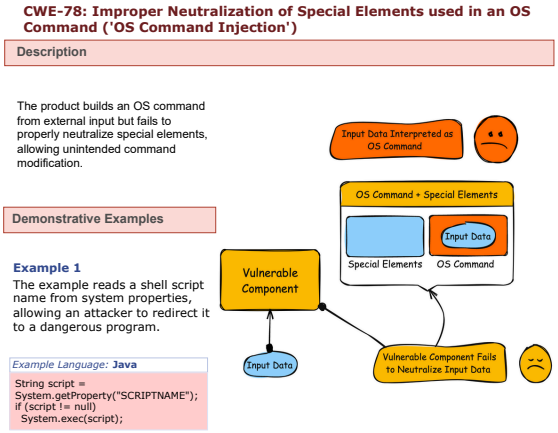


Figure 3: An example of CWE-78 on the MITRE website.

### 3.1 CWE Seeds Collection

First, we collect Common Weakness Enumeration (CWE) documents from the MITRE. The MITRE is an official website that serves as an authoritative resource for understanding and analyzing CWEs. Each CWE entry on the website includes a comprehensive description, highlighting the nature of the vulnerability, its potential impact, and common scenarios in which it arises. Additionally, the entries feature illustrative code examples that contrast vulnerable and secure implementations, offering both context and practical insights. As shown in Figure 3, CWE-78 (OS Command Injection) provides such insights. Secure-Instruct leverages the information from the MITRE website for each CWE, namely 1) the category of CWE, 2) the overall description, and 3) code snippet examples with their explanations. In total, we collected 44 CWEs from MITRE.

### 3.2 Synthetic Data Generation By Prompting LLMs

Given seed CWE documents, Secure-Instruct employs two data generation strategies to prompt LLM for generating instruction-tuning datasets, namely vulnerable-code driven (Secure-Instruct$_{(vul,\ secure)}$) and secure-code driven (Secure-Instruct$_{secure}$).

**Secure-Instruct$_{(vul,\ secure)}$.** As Figure 2 illustrates, the vulnerable-code driven data generation includes four steps: (1) prompting an LLM (e.g., GPT 4o) to generate vulnerable code given one CWE information; (2) verifying that the LLM-generated code indeed contains the specified CWE; (3) prompting LLMs to fix the vulnerabilities and produce secure code; and (4) confirming the secure code is vulnerability-free. Vulnerable-code driven generation ensures the success of generating both vulnerable and secure code for specific CWEs. We employ two powerful LLMs for code generation, i.e., ChatGPT (gpt-4o-2024-08-06) and Claude (claude-3-5-sonnet-20241022), to generate intentionally vulnerable code snippets. The templates for generating and fixing vulnerable code are included in our replication package. For each pair of "CWE" and "programming language", Secure-Instruct$_{(vul,\ secure)}$ takes both the CWE description and a randomly selected code example from the seed corpus and prompts each LLM to generate 10 vulnerable code snippets. In total, Secure-Instruct$_{(vul,\ secure)}$ collected 1,560 code snippets spanning 78 CWE-language pairs. Then, Secure-Instruct$_{(vul,\ secure)}$ utilizes static security analyzers to verify the vulnerabilities present in the generated code. Finally, we instruct the LLMs to fix the vulnerable code and verify that the vulnerabilities are gone using static security analyzers. Since static analysis tools can produce both false positives and false negatives, we use two static security analyzers, CodeQL (GitHub 2023) and SonarQube (SonarQube 2025), to complement each other in verification. A code snippet is considered vulnerable only if both tools produce identical detection results.

After verification of the 1,560 snippets by static analyzers, 475 were confirmed to be truly vulnerable. These verified snippets then served as input seeds for the LLMs to generate five fixes for each snippet. Finally, static analyzers are used again to verify whether the generated fixes were valid. After generating the dataset through our pipeline and verifying it using static analyzers, we performed a de-duplication step to eliminate potential overlaps with evaluation benchmarks. This ensures that our evaluation remains fair and unbiased. The details of this de-duplication process are described in Section 4. In the end, Secure-Instruct$_{(vul,\ secure)}$ produced 856 verified (vulnerable, fixed) pairs across 25 CWEs and 6 languages, covering 39 CWE-language combinations.

**Secure-Instruct$_{secure}$.** Secure-Instruct$_{(vul,\ secure)}$'s double verification process results in many LLM-generated code snippets being filtered out, e.g., LLMs may fail to generate vulnerable code or fix vulnerable code in many cases, leading to a limited set of secure code snippets. To source an even larger set of secure code snippets, we propose an alternative approach, Secure-Instruct$_{secure}$. Instead of generating vulnerable code and then repairing it, we directly instruct LLMs to produce secure code using CWE documents as prompts. For each CWE-language pair, we generate 100 secure code snippets that pass verification by both CodeQL and SonarQube, ensuring they do not contain vulnerabilities. Similarly, we performed a de-duplication step to make sure that all generated code snippets are not duplicated with evaluation benchmarks.

### 3.3 Instruction Tuning LLMs for Secure Code Generation

Secure-Instruct utilizes LLMs to generate a problem description as an "instruction" for each of the instruction-tuning datasets. Then Secure-Instruct employs two instruction tuning methods, i.e., 1). Standard instruction tuning and 2). Secure instruction tuning.

**Generating Instruction.** We follow the work (He et al. 2024) to construct the instruction for each code snippet using LLMs. After that, we format the instruction-tuning datasets by putting the code snippet in the <response> tag and the generated instructions in the <instruction> tag, following a prior work (Taori et al. 2023).

**Standard Instruction Tuning.** This process fine-tunes an LLM on instruction-response pairs to improve its ability to follow natural language instructions. It uses supervised fine-tuning (SFT), where the model learns by minimizing loss over the provided dataset, which has been adopted by many prior studies (Wei et al. 2024; Xu et al. 2024; Luo et al. 2023).

**Secure-Specialized Instruction Tuning.** SafeCoder (He et al. 2024) proposed a specialized instruction-tuning technique for secure code generation by leveraging pairs of vulnerable code and fixed code mined from open-sourced software repositories. Since we collected 856 pairs of vulnerable and from Secure-Instruct$_{(vul,\ secure)}$, we applied this instruction tuning approach to our dataset. Prior research (He and Vechev 2023) highlights a critical concern: exclusively using secure code examples for instruction-tuning can lead to a decline in code functionality. To address this limitation, we follow the work (He et al. 2024) via the integration of a secure code dataset and a functional code dataset during instruction-tuning. It uses standard instruction tuning for the functional dataset and secure instruction tuning for the secure dataset to improve both functional and secure performance for LLMs. Specifically, we incorporate the functional code dataset from the Evol-Instruct dataset (Luo et al. 2023), a high-quality dataset designed to enhance the functional robustness and generalization of code generation models.

## 4 Evaluation

### 4.1 Evaluation Setup

**Subject Large Language Models.** We evaluate Secure-Instruct to improve four representative small and medium-sized LLMs with 1 to 7 billion parameters: CodeGen2-7B (Nijkamp et al. 2023), CodeLlama-7B (Roziere et al. 2023), Mistral-7B (Jiang et al. 2023), and StarCoder-1B (Li et al. 2023). These models are specifically designed for code

| Attribute | Secure-Instruct (vul, secure) | Secure-Instruct secure |
|---|---|---|
| # of Code Snippets | 856 | 15,600 |
| # of Average LOCs | 52.0 | 50.0 |
| # of CWEs | 25 | 44 |
| # of CWE-language pairs | 39 | 78 |

Table 1: The statistics of the fine-tuning datasets used in Secure-Instruct.

generation, offering a balance between computational efficiency and performance. These models are widely used in prior works on secure code generation, such as (He and Vechev 2023; He et al. 2024; Wei et al. 2024) for a fair comparison. For each of the four studied LLMs, we experimented with two settings of Secure-Instruct: (1) applying security-specialized instruction tuning on the dataset of pairs of vulnerable and secure code, i.e., from Secure-Instruct$_{(vul,\ secure)}$; and (2) applying standard instruction tuning on the secure code snippets from Secure-Instruct$_{secure}$. Table 1 summarizes the statistics of the resulting fine-tuning datasets using Secure-Instruct.

**Details on Instruction-Tuning LLMs.** We use the parameter-efficient technique, LoRA (Hu et al. 2021), to instruction-tune the 7B LLMs. LoRA reduces the GPU memory usage and maintains a promising performance. For the 1B model, we apply full-weight instruction-tuning. All experiments are conducted on five Nvidia RTX A6000 GPUs with 48GB of memory. For full-weight instruction-tuning, we set the learning rate to 2e-5, while for LoRA fine-tuning, the learning rate is 3e-4. Each model is tuned in 2 epochs. In LoRA, we configure the LoRA alpha to 16 to control the scaling of LoRA layers and set the dropout rate to 0.05.

**Evaluation Benchmarks.** We evaluate Secure-Instruct using two secure-code generation benchmarks: (1) *CWEBench* crafted and extended by us, and (2) *CWEval* (Peng et al. 2025), which enables validation of both functional correctness and security by test cases.

First, we craft *CWEBench* by extending prior benchmarks for secure code generation (Siddiq and Santos 2022; Pearce et al. 2022; He et al. 2024). Such prior benchmarks are composed of security-specific scenarios and are sourced from three resources, i.e., MITRE documentation (CWE-MITRE 2022), CodeQL repository (GitHub 2023), and scenarios manually crafted by the authors. We first combined all three existing benchmarks (Siddiq and Santos 2022; Pearce et al. 2022; He et al. 2024), removed duplicates, and excluded the scenarios derived from MITRE examples, as Secure-Instruct uses MITRE documents as seeds for LLMs to generate fine-tuning datasets. We define a **secure ratio** to measure LLM's capability in CWEBench. *Secure ratio* is the percentage of secure code identified by CodeQL among all the generated code. To ensure robust results, we generated 100 samples for each security-specific scenario with a temperature of 0.8. A piece of LLM-generated code is deemed *secure code* if CodeQL detects no vulnerabilities in the code. Then, we followed a similar process as prior work (He et al. 2024) to design 36 new security-specific scenarios covering 27 CWEs by adapting examples from the CodeQL repository. In the end, our crafted *CWEBench* consists of 93 security-specific scenarios covering 44 CWEs in six languages.

Second, we include a recent benchmark *CWEval* (Peng et al. 2025), which uses test cases for automated verification of both *functional correctness* and *security* without relying on static security checkers. In addition, *CWEval* provides manually crafted comprehensive specifications that describe complex user intentions and logic, which means it is a much closer step to real-world applications. In total, *CWEval* consists of 119 security scenarios covering 31 CWE types in five programming languages. *CWEval* comes with three metrics, namely **Func@k**, **Sec@k** and **Func-Sec@k** to evaluate the security and functionality of generated code. In this study, we set k as 1. Specifically, *Func@1* measures the percentage of security scenarios whose top-1 generated code is functionally correct (i.e., passes all functional test cases). *Sec@1* is defined as the percentage of security scenarios whose top-1 solution is secure (i.e., passes all security test cases). *Func-Sec@1* concerns both functional correctness and security.

**De-duplication between fine-tuning datasets and evaluation benchmarks.** To prevent data leakage between the fine-tuning dataset and the two benchmark datasets, we performed de-duplication between training and evaluation following (Zhou et al. 2025). Specifically, we used MinHash with Locality-Sensitive Hashing (LSH) to detect potential duplicates. Code snippets were grouped based on similarity, and pairs with a Jaccard similarity above 0.7 were flagged as potential duplicates. After that, we removed these flagged snippets from the synthesized dataset.

## 4.2 Evaluation Results

In this section, we present our evaluation results in both secure and functional benchmarks.

**Results of CWEBench.** We evaluated the pre-trained models and instruction-tuned models by Secure-Instruct. We also conducted a comparative analysis between Secure-Instruct and SafeCoder (He et al. 2024). In particular, we reused the two released models, i.e., CodeLlma-7B and Mistral-7B, which were already instruction-tuned by SafeCoder. Table 2 presents the results on *CWEBench*. The highest secure ratio is highlighted in bold. Across all models, Secure-Instruct$_{(vul,\ secure)}$ and Secure-Instruct$_{secure}$ demonstrate significant improvements in secure code generation. Compared with the pre-trained models, Secure-Instruct achieved the highest secure ratio across all models. For example, CodeLlama-7B achieved the highest secure ratio among the four models when instruction-tuned with Secure-Instruct$_{secure}$, improving by 46.6% (the secure ratio from 47.6% to 69.8%), compared to an increase 35.3% under Secure-Instruct$_{(vul,\ secure)}$. The secure ratio of CodeGen2-7B is improved from 48.8% to 57.0% of secure ratio by Secure-Instruct$_{secure}$ and to 49.0% by Secure-Instruct$_{(vul,\ secure)}$. Mistral-7B achieved an increase of 28.9% by Secure-Instruct$_{secure}$. StarCoder-1B also achieves a notable improvement, i.e., secure ratio rising from 55.4%

| | Code Llama-7B | Code Gen2-7B | Mistral-7B | Star Coder-1B |
|---|---|---|---|---|
| Pre-trained | 47.6% | 48.8% | 51.5% | 55.4% |
| SafeCoder | 60.6% (↑ 27.3%) | NA | 60.4% (↑ 17.3%) | NA |
| **Secure-Instruct (vul,secure)** | 64.4% (↑ 35.3%) | 49.0% (↑ 0.4%) | 61.3% (↑ 19.0%) | 63.4% (↑ 14.4%) |
| **Secure-Instruct secure** | **69.8% (↑ 46.6%)** | **57.0% (↑ 16.8%)** | **66.4% (↑ 28.9%)** | **67.5% (↑ 21.8%)** |

Table 2: The secure ratio for pre-trained and instruction-tuned models using Secure-Instruct and SafeCoder on CWEBench. ↑ and ↓ indicate relatively change from pre-trained models. NA means the model not released by SafeCoder.

to 67.5%. Across the four LLMs, Secure-Instruct improves them to generate more secure code, and Secure-Instruct$_{secure}$ shows consistently better results than Secure-Instruct$_{(vul,\ secure)}$.

Compared to SafeCoder, our two models consistently outperform the SafeCoder models, with +15.2% in CodeLlama-7B and +9.9% in Mistral-7B using Secure-Instruct$_{secure}$. At the CWE category level, CodeLlama-7B instruction tuned with Secure-Instruct$_{secure}$ achieves the highest or ties for highest secure ratio in 29 of 44 CWEs, outperforming both its pre-trained baseline and the SafeCoder-tuned variant. Similarly, Mistral-7B with Secure-Instruct$_{secure}$ leads in 25 of 44 CWEs relative to its pretrained and SafeCoder-tuned counterpart. Due to the limited space, we put all detailed analyses in our replication package. ***These results demonstrate that Secure-Instruct$_{secure}$ significantly enhances secure code generation, outperforming the SOTA approach, SafeCoder***.

**Results of CWEval.** Table 3 presents the results on the benchmark *CWEval*. Similar to *CWEBench*, Secure-Instruct achieves the highest *Func-Sec@1* across all evaluated LLMs. Notably, CodeLlama-7B and Mistral-7B show significant improvements, with increases of 157.3% and 46.4%, compared to their pre-trained versions. For *Sec@1*, CodeLlama-7B, CodeGen2-7B, and StarCoder-1B achieved the highest secure ratio using Secure-Instruct, with 33.7%, 27.9%, and 16.4%, respectively. The results of *Func@1* are put in our replication package due to limited space.

Interestingly, SafeCoder exhibits a performance drop on CWEval. For example, SafeCoder's CodeLlama-7B achieves only 7.1% *Func-Sec@1*, which is lower than the pre-trained model. A similar decline is observed in SafeCoder's Mistral-7B. These findings are consistent with the observations reported in the study (Peng et al. 2025). ***Overall, LLMs with Secure-Instruct consistently outperform both their pre-trained versions and SafeCoder models in secure code generation across CWEBench and CWEval.***

**Results of Functional Benchmarks.** We evaluated our approaches with functional benchmarks, i.e., HumanEval and MBPP, two widely used benchmarks for assessing code generation in Python. For each task, we generate 40 samples and assess LLMs' functionality using the *Pass@k* metric, including *Pass@1*. *Pass@k* is a standard metric for evaluating code generation, which represents the probability that at least one of the $k$ generated candidates for a given problem is correct. Table 4 presents the Pass@1 (Pass@10 is available in our replication package) on the HumanEval and MBPP benchmarks for pre-trained LLMs, instruction-tuned LLMs by SafeCoder and Secure-Instruct. For CodeLlama-7B, using Secure-Instruct$_{secure}$ achieves the highest *Pass@1* on HumanEval at 35.6%, outperforming both the pre-trained and SafeCoder versions. For Mistral-7B, the model using the Secure-Instruct$_{(vul,\ secure)}$ achieves the highest score on MBPP at 30.7%. Codegen2-7B achieved the highest *Pass@1* using Secure-Instruct$_{secure}$, with 16.3% on HumanEval and 22.8% on MBPP. Overall, most models show performance improvements on both datasets, except for StarCoder-1B.

## 5 Discussions

**The accuracy of CodeQL.** In CWEBench, the ground-truth of security is based on CodeQL, which is a static tool and may produce false positives. To address this concern, we randomly sampled 291 code snippets labeled as secure by CodeQL from four models and manually verified their security. All sampled snippets were confirmed to be truly secure. To further support our conclusion, we used an additional benchmark, CWEval, which evaluates security through test cases.

**The similarity between Secure-Instruct's instruction-tuning datasets and our evaluation benchmark.** We performed a similarity analysis between the instruction-tuning data and benchmark data. Specifically, we first extracted all the secure code examples from the CodeQL repository for the 44 CWEs of our benchmark. We compare each LLM-generated secure code with the secure code examples from the CodeQL repository under the same CWE. We then calculated the cosine similarity of each pair using TF-IDF embeddings (Sparck Jones 1972). Figure 5 depicts the average cosine similarity score for the data generated by Secure-Instruct$_{secure}$, which is 0.17, while the average score for data generated by Secure-Instruct$_{(vul,\ secure)}$ is 0.23. Secure-Instruct$_{secure}$'s values align with findings from previous work (Wei et al. 2024) on synthesizing instruction-tuning data using LLMs, which reported similarity scores typically ranging between 0.1 and 0.2 in the previous synthesis techniques. Importantly, this result indicates that the improvements from Secure-Instruct are not simply due to the inclusion of data from the same distribution of the CodeQL repository. Instead, the generated data provides additional diversity for the instruction-tuning dataset.

**Diversity of Secure-Instruct's synthetic datasets.** We followed a similar procedure to compute the pair-wise similarity for Secure-Instruct$_{(vul,\ secure)}$ and Secure-Instruct$_{secure}$ using TF-IDF. The similarity of Secure-Instruct$_{(vul,\ secure)}$ is 0.17. The similarity of Secure-Instruct$_{secure}$ is 0.16. Results show that datasets from both approaches are basically diverse.

| | CodeLlama-7B | | CodeGen2-7B | | Mistral-7B | | StarCoder-1B | |
|---|---|---|---|---|---|---|---|---|
| | Sec@1 | Func-Sec@1 | Sec@1 | Func-Sec@1 | Sec@1 | Func-Sec@1 | Sec@1 | Func-Sec@1 |
| Pre-trained | 17.8% | 8.9% | **17.9%** | 7.4% | 20.4% | 12.5% | 16.2% | 6.1% |
| SafeCoder | 17.4% (↓ 2.2%) | 7.1% (↓ 20.2%) | NA | NA | 17.4% (↓ 14.7%) | 11.5% (↓ 8.0%) | NA | NA |
| **Secure-Instruct (vul,secure)** | 31.2% (↑ 75.3%) | **22.9% (↑ 157.3%)** | 17.7% (↓ 1.1%) | 7.2% (↓ 2.7%) | 25.5% (↑ 25.0%) | 17.9% (↑ 43.2%) | **16.4% (↑ 1.2%)** | **7.4% (↑ 21.3%)** |
| **Secure-Instruct secure** | **33.7% (↑ 89.3%)** | **22.9% (↑ 157.3%)** | 16.4% (↓ 8.4%) | **7.5% (↑ 1.4%)** | **27.9% (↑ 36.8%)** | **18.3% (↑ 46.4%)** | 15.5% (↓ 4.3%) | **7.4% (↑ 21.3%)** |

Table 3: Sec@1 and Func-Sec@1 results on CWEval for pre-trained LLMs and LLMs tuned by SafeCoder and Secure-Instruct (ours).

| | CodeLlama-7B | | CodeGen2-7B | | Mistral-7B | | StarCoder-1B | |
|---|---|---|---|---|---|---|---|---|
| | HumanEval | MBPP | HumanEval | MBPP | HumanEval | MBPP | HumanEval | MBPP |
| Pre-trained | 25.5% | 28.3% | 15.3% | 20.4% | 23.0% | 24.7% | **11.7%** | 13.7% |
| SafeCoder | 32.0% (↑ 25.5%) | 33.7% (↑ 19.1%) | NA | NA | 31.3% (↑ 36.1%) | 29.3% (↑ 18.6%) | NA | NA |
| **Secure-Instruct (vul,secure)** | 34.7% (↑ 36.1%) | **36.0% (↑ 27.2%)** | 8.1% (↓ 47.1%) | 15.4% (↓ 24.5%) | **31.6% (↑ 37.4%)** | **30.7% (↑ 24.3%)** | 6.7% (↓ 42.7%) | 12.2% (↓ 10.9%) |
| **Secure-Instruct secure** | **35.6% (↑ 39.6%)** | 34.7% (↑ 22.6%) | **16.3% (↑ 6.5%)** | **22.8% (↑ 11.8%)** | 25.9% (↑ 12.6%) | 27.1% (↑ 9.7%) | 10.9% (↓ 6.8%) | **17.7% (↑ 29.2%)** |

Table 4: Pass@1 results on HumanEval and MBPP for pre-trained LLMs and LLMs tuned by SafeCoder-tuned and Secure-Instruct.

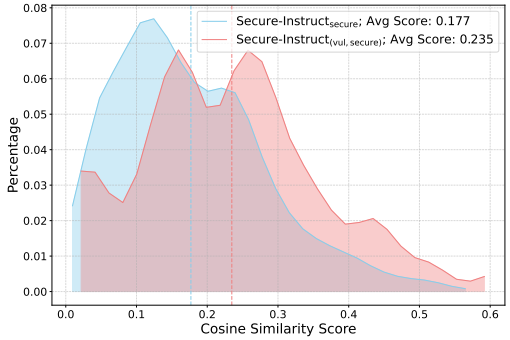


Figure 4: Cosine similarities between Secure-Instruct's instruction-tuning dataset and the secure code examples by CodeQL (i.e., our evaluation benchmark).

**Low cost of Secure-Instruct in generating instruction-tuned datasets.** Secure-Instruct generates high-quality instruction-tuning data with low cost and no human effort. Using ChatGPT (GPT-4o-2024-08-06), the total cost for generating all secure code snippets is USD 114.09. Similarly, the cost is USD 109.74 for Claude. The cost per CWE-language is USD 2.87. In contrast, SafeCoder (He et al. 2024) faces scalability challenges, filtering 145M commits down to just 1,200 code pairs, and requiring manual selection of 465 high-quality examples. Secure-Instruct eliminates such overhead, offering a faster, cheaper, and automated alternative for generating secure and reliable datasets.

**Secure-Instruct$_{(vul, secure)}$ v.s. Secure-Instruct$_{secure}$.** Secure-Instruct$_{secure}$ uses a much larger dataset than Secure-Instruct$_{(vul, secure)}$ (15.6K vs. 856), so its better performance is expected. To compare fairly, we downsampled Secure-Instruct$_{secure}$ to 865 examples, covering all CWE-language pairs. Table 5 shows the comparison results across the four LLMs. Notably, the differences between the two are minimal, indicating essentially equivalent effectiveness. This shows that Secure-Instruct$_{secure}$ still generates highly effective synthetic data without being driven by vulnerable code first.

| | Code Llama-7B | Code Gen2-7B | Mistral-7B | Star Coder-1B |
|---|---|---|---|---|
| **Secure-Instruct$_{(vul, secure)}$** | **64.4%** | 49.0% | 61.3% | 63.4% |
| **Secure-Instruct$_{secure}$** | 61.9% | **49.1%** | **62.0%** | **66.2%** |

Table 5: The secure ratios of Secure-Instruct$_{(vul, secure)}$ and Secure-Instruct$_{secure}$ with the same size (865) of training dataset.

## 6 Conclusion

We presented Secure-Instruct, a novel framework for improving secure code generation by leveraging LLMs to synthesize instruction-tuning datasets. Secure-Instruct overcomes the key limitations of the SOTA technique, particularly addressing dataset imbalance, inefficient data curation, and the significant manual effort required. Unlike existing approaches, Secure-Instruct can be easily adapted to support new CWEs. We also introduce CWEBench, the

largest benchmark for secure code generation. Our evaluation shows that Secure-Instruct significantly improves the performance of four LLMs, achieving an average improvement of 28.5% on CWEBench and a 140.8% improvement of Func-Sec@1 on CWEval. Results show that Secure-Instruct outperforms the SOTA method, demonstrating both scalability and efficiency.

## References


Chen, M.; Tworek, J.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

CWE-MITRE. 2022. Common Weakness Enumeration. https://cwe.mitre.org/index.html. Accessed: July 30, 2025.

Fu, Y.; Liang, P.; Tahir, A.; Li, Z.; Shahin, M.; Yu, J.; and Chen, J. 2023. Security weaknesses of copilot generated code in github. *arXiv preprint arXiv:2310.02059*.

GitHub. 2023. CodeQL. https://codeql.github.com. Accessed: 2025-06-30.

GitHub. 2024. GitHub Copilot - Your AI pair programmer. https://github.blog/news-insights/product-news/github-copilot-x-the-ai-powered-developer-experience/. Accessed: 2024-10-14.

Gong, J.; Duan, N.; Tao, Z.; Gong, Z.; Yuan, Y.; and Huang, M. 2024. How Well Do Large Language Models Serve as End-to-End Secure Code Producers? *arXiv preprint arXiv:2408.10495*.

Hajipour, H.; Hassler, K.; Holz, T.; Schönherr, L.; and Fritz, M. 2023. CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. *arXiv preprint arXiv:2302.04012*.

Hajipour, H.; Schönherr, L.; Holz, T.; and Fritz, M. 2024. HexaCoder: Secure Code Generation via Oracle-Guided Synthetic Training Data. *arXiv preprint arXiv:2409.06446*.

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

He, J.; Vero, M.; Krasnopolska, G.; and Vechev, M. 2024. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jiang, A. Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D. S.; Casas, D. d. l.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825*.

Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; and Svyatkovskiy, A. 2023. Inferfix: End-to-end program repair with llms. In *FSE*, 1646–1656.

Joshi, H.; Sanchez, J. C.; Gulwani, S.; Le, V.; Verbruggen, G.; and Radiček, I. 2023. Repair is nearly generation: Multilingual program repair with llms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4): 5131–5140.

Khoury, R.; Avila, A. R.; Brunelle, J.; and Camara, B. M. 2023. How secure is code generated by chatgpt? In *SMC 2023*, 2445–2451. IEEE.

Li, J.; Rabbi, F.; Cheng, C.; Sangalay, A.; Tian, Y.; and Yang, J. 2024a. An Exploratory Study on Fine-Tuning Large Language Models for Secure Code Generation. *arXiv preprint arXiv:2408.09078*.

Li, J.; Sangalay, A.; Cheng, C.; Tian, Y.; and Yang, J. 2024b. Fine Tuning Large Language Model for Secure Code Generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 86–90.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Lin, F.; Kim, D. J.; Tse-Husn; and Chen. 2024. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. arXiv:2403.15852.

Liu, Z.; Tang, Y.; Luo, X.; Zhou, Y.; and Zhang, L. F. 2024. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*.

Longpre, S.; Hou, L.; Vu, T.; Webson, A.; Chung, H. W.; Tay, Y.; Zhou, D.; Le, Q. V.; Zoph, B.; Wei, J.; et al. 2023. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, 22631–22648. PMLR.

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

MITRE. 2025. Common Weakness Enumeration (CWE). https://cwe.mitre.org/. Accessed: 2025-04-15.

Mohsin, A.; Janicke, H.; Wood, A.; Sarker, I. H.; Maglaras, L.; and Janjua, N. 2024. Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs. *arXiv preprint arXiv:2406.12513*.

Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; and Zhou, Y. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *ICLR*.

OpenAI. 2024. ChatGPT. https://chat.openai.com/. Large language model.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *S&P*, 754–768. IEEE.

Peng, J.; Cui, L.; Huang, K.; Yang, J.; and Ray, B. 2025. CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. *arXiv preprint arXiv:2501.08200*.

Perry, N.; Srivastava, M.; Kumar, D.; and Boneh, D. 2023. Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2785–2799.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Garg, S.; and Dolan-Gavitt, B. 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)*, 2205–2222.

Siddiq, M. L.; and Santos, J. C. 2022. 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*, 29–33.

Siddiq, M. L.; and Santos, J. C. 2023. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889*.

SonarQube. 2025. SonarQube Static Code Analysis.

Sparck Jones, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1): 11–21.

Taori, R.; Gulrajani, I.; Zhang, T.; Dubois, Y.; Li, X.; Guestrin, C.; Liang, P.; and Hashimoto, T. B. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

Tihanyi, N.; Bisztray, T.; Ferrag, M. A.; Jain, R.; and Cordeiro, L. C. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2): 1–42.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wang, J.; Cao, L.; Luo, X.; Zhou, Z.; Xie, J.; Jatowt, A.; and Cai, Y. 2023. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. *arXiv preprint arXiv:2310.16263*.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and Zhang, L. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Xu, C.; Sun, Q.; Zheng, K.; Geng, X.; Zhao, P.; Feng, J.; Tao, C.; Lin, Q.; and Jiang, D. 2024. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Zhou, X.; Weyssow, M.; Widyasari, R.; Zhang, T.; He, J.; Lyu, Y.; Chang, J.; Zhang, B.; Huang, D.; and Lo, D. 2025. LessLeak-Bench: A First Investigation of Data Leakage in LLMs Across 83 Software Engineering Benchmarks. *arXiv preprint arXiv:2502.06215*.

# A Appendix A: Prompt Template in Data Synthesizing

In our study, we leverage two schemes to synthesize data by vulnerable-code driven for Secure-Instruct$_{(vul,\ secure)}$ and secure-code driven for Secure-Instruct$_{(vul,\ secure)}$. In Secure-Instruct$_{(vul,\ secure)}$, Figure 5 shows the prompt template that we first prompt LLMs to generate vulnerable code snippets. Then another prompt is used for vulnerable code fixing, shown in Figure 6.

In Secure-Instruct$_{secure}$, the prompt shown in Figure 7 is similar to the first step of Secure-Instruct$_{(vul,\ secure)}$, and the only change is that we directly ask LLMs to generate secure code snippets.

**Prompt Template for Vulnerable Code Generation of Secure-Instruct$_{(vul,\ secure)}$**

**Task:** Please generate an easily understandable **vulnerable** code snippet based on the following description of CWE. (Please consider the diversity of the **vulnerable** code examples for each generation.)

**CWE:** {<CWE_OVERALL_DESCRIPTION>}

**A Vulnerable Code Example:** This is {<LANGUAGE>} language example.

{<CODE>}

**Explanation of the Example:** {<EXPLANATION>}

**Can you generate a vulnerable code example for** {<TARGET_LANGUAGE>} **language?**

Figure 5: Prompt template used for vulnerable code generation in Secure-Instruct$_{(vul,\ secure)}$.

**Prompt Template for Fixing Vulnerable Code of Secure-Instruct$_{(vul,\ secure)}$**

**Task:** Please fix the vulnerable code based on the following description of CWE:

**CWE:** {<CWE_OVERALL_DESCRIPTION>}

**The Vulnerable Code:**

{<GENERATED_VULNERABLE_CODE>}

**Can you FIX the code for** {<LANGUAGE>} **? Please make sure the code is secure to the CWE mentioned above and runnable.**
**Task:** Please generate an easily understandable secure code snippet based on the following description of CWE. (Please consider the diversity of the vulnerable code examples for each generation.)

Figure 6: Prompt template used for vulnerable code fixing in Secure-Instruct$_{(vul,\ secure)}$.

**Prompt Template for Secure Code Generation of Secure-Instruct$_{secure}$**

**Task:** Please generate an easily understandable **secure** code snippet based on the following description of CWE. (Please consider the diversity of the **secure** code examples for each generation.)

**CWE:** {<CWE_OVERALL_DESCRIPTION>}

**A Vulnerable Code Example:** This is {<LANGUAGE>} language example.

{<CODE>}

**Explanation of the Example:** {<EXPLANATION>}

**Can you generate a secure code example for** {<TARGET_LANGUAGE>} **language?**

Figure 7: Prompt template used for secure code generation in Secure-Instruct$_{secure}$.

**Prompt Template for Instruction Generation**

Create a single, very short (maximum two sentences) non-detailed functionality description that could be used as a prompt to generate either of the code snippets below...
{<The_Non-vulnerable_code_snippet>}

Figure 8: Prompt template used for instruction generation.

# B Appendix B: Distribution of CWEBench

Figure 9 shows the distribution of evaluated scenarios for languages and CWEs. Among 93 scenarios, 19 are for C, 9 are for Go, 14 are for Java, 18 are for JavaScript, 22 are for Python, and 11 are for Ruby. These scenarios cover 44 distinct CWEs, enabling a comprehensive evaluation.

# C Appendix C: Detailed results of CWEBench

Table 6 breaks down the detailed results on CWEBench of CodeLlama-7B and Mistral-7B in their pre-trained, SafeCoder, and Secure-Instruct, on each CWE in our evaluation benchmark. The highest secure ratio for each CWE is highlighted in bold. In the scenarios whose CWEs are covered in both approaches (the top section of the table), our analysis shows that CodeLlama-7B fine-tuned with Secure-Instruct$_{secure}$ achieved the highest overall secure ratio (70.6%), outperforming SafeCoder by 2.3% in secure ratio, Mistral-7B fine-tuned with Secure-Instruct$_{secure}$ achieved 68.9% secure ratio, with a 3.7% higher than SafeCoder. In the generalization evaluation (i.e., with the full security-specific scenarios of our benchmark), Secure-Instruct$_{secure}$demonstrates strong generalization capabilities across a broader range of CWEs for both CodeLlama-7B and Mistral-7B. Specifically, CodeLlama-7B with Secure-Instruct$_{secure}$ significantly outperforms SafeCoder, achieving a 69.2% secure ratio, while Mistral-7B achieves a 66.4% secure ratio.

# D Appendix D: Limitations

LLMs can produce different outputs for the same prompt. To prevent the huge variations, we set a large (100) sample size to ensure the stabilization of results. CWEBench relies on CodeQL to detect vulnerabilities. Such a static analyzer may produce false positives. Although we manually reviewed samples, misclassifications could still influence the conclusions. We performed de-duplication across both our fine-tuning dataset and evaluation benchmarks (CWEBench and CWEval). However, this process may fail to detect duplicates, leading to false negatives.

We follow prior work and choose LLMs with smaller parameter sizes to make fine-tuning feasible under limited computational resources. Instruction-tuning larger models, such as ChatGPT-4o, would incur significantly higher costs. As a result, we acknowledge that our findings may not fully generalize to larger LLMs, and we consider this a direction for future investigation. We include 44 distinct CWEs in this study, reflecting a broad range of vulnerabilities. However, numerous additional CWEs remain unaddressed in this study. Moreover, individual CWE can manifest in various real-world scenarios, further highlighting the complexity of comprehensive evaluation. To improve coverage, we expanded our security-sensitive prompts to incorporate additional CWEs beyond those included in earlier studies. Compared to prior work (Pearce et al. 2022; He and Vechev 2023; He et al. 2024; Li et al. 2024a), our study evaluates a significantly greater number of CWEs, providing a more thorough assessment of generated code vulnerabilities. Future efforts should include broader CWEs. Additionally, we include six programming languages in our evaluation benchmark. While these languages are highly popular and widely adopted in software development, our findings may not generalize to all programming languages. Future work should expand the language set to strengthen the robustness of the results.

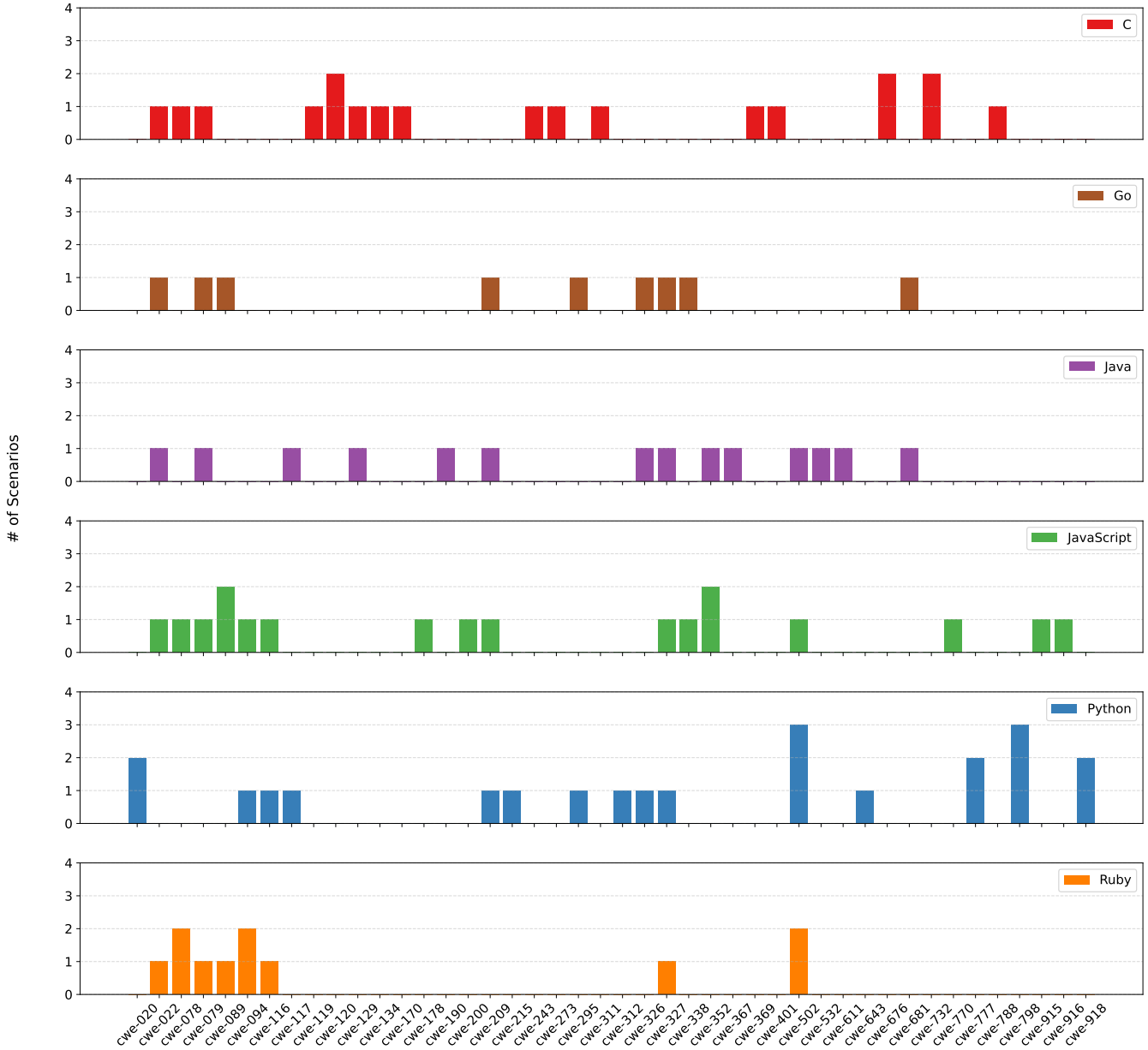


Figure 9: The distribution of languages and CWEs in CWEBench.

Table 6: The results of secure ratio in CodeLlama-7B and Mistral-7B. The top section lists CWEs that are included in the fine-tuning dataset of SafeCoder, while the bottom lists CWEs that are not covered in the SafeCoder dataset.

| CWE | CodeLlama-7B | | | | Mistral-7B | | | |
|---|---|---|---|---|---|---|---|---|
| | Pre-trained | SafeCoder | Secure-Instruct Secure | Secure-Instruct (Vul, Secure) | Pre-trained | SafeCoder | Secure-Instruct Secure | Secure-Instruct (Vul, Secure) |
| cwe-022 | 16.0 | **43.2** | 32.0 | 37.2 | 14.8 | 34.6 | **38.0** | 20.8 |
| cwe-078 | 16.8 | **79.0** | 50.7 | 40.8 | 30.5 | **64.2** | 60.0 | 52.0 |
| cwe-079 | 37.8 | **84.6** | 64.8 | 67.0 | 29.2 | **68.8** | 68.6 | 48.6 |
| cwe-089 | 52.0 | 70.2 | **82.5** | 72.5 | 51.7 | 60.8 | 67.0 | **69.0** |
| cwe-116 | 66.7 | 65.0 | 69.7 | **75.3** | 65.7 | 68.7 | **71.3** | 65.3 |
| cwe-119 | 87.0 | **99.0** | 67.0 | 47.0 | 93.0 | **100.0** | 5.0 | 96.0 |
| cwe-190 | **24.0** | 7.0 | 23.0 | 23.0 | **25.0** | 1.0 | 5.0 | 15.0 |
| cwe-200 | 24.0 | **99.0** | **99.0** | 17.0 | 34.0 | **98.0** | 73.0 | 50.0 |
| cwe-295 | 10.5 | 78.5 | **82.0** | 78.0 | 17.5 | 78.5 | 78.0 | **85.0** |
| cwe-326 | 40.0 | 33.3 | 83.0 | **98.0** | 55.7 | **72.7** | 59.3 | 60.3 |
| cwe-327 | 47.8 | **68.4** | 64.0 | 64.0 | 61.2 | 54.6 | **77.0** | 74.8 |
| cwe-338 | 38.5 | 42.0 | 56.0 | **71.0** | 13.5 | 31.5 | **50.5** | 46.0 |
| cwe-352 | 87.7 | 93.3 | 95.7 | **100.0** | 94.7 | 85.7 | **97.3** | 96.3 |
| cwe-502 | 58.3 | 79.1 | **97.3** | 80.7 | 68.3 | 89.4 | **93.9** | 80.0 |
| cwe-611 | 4.0 | **54.0** | 40.0 | 0.0 | 0.0 | **51.0** | 40.0 | 0.0 |
| cwe-676 | 38.0 | 48.5 | 47.5 | **49.5** | 56.0 | 54.5 | **93.0** | 53.5 |
| cwe-681 | **58.0** | 50.5 | 56.5 | 54.5 | 54.0 | 35.5 | **80.5** | 50.5 |
| cwe-732 | 32.0 | 75.0 | **99.5** | 91.0 | 57.0 | 89.5 | **93.5** | 92.5 |
| cwe-915 | 24.0 | 82.0 | 95.0 | **97.0** | 25.0 | 51.0 | 37.0 | **77.0** |
| cwe-916 | 94.0 | 97.0 | **100.0** | 98.0 | 96.0 | 95.0 | 99.0 | **100.0** |
| Total (above) | 43.4 | 67.9 | **70.4** | 66.3 | 48.2 | 64.9 | **69.4** | 62.3 |
| cwe-020 | **75.0** | 50.5 | **75.0** | 61.0 | 61.0 | 75.0 | **78.0** | 54.0 |
| cwe-094 | 88.0 | 84.2 | **99.2** | 98.0 | 88.8 | 82.0 | 91.5 | **97.2** |
| cwe-117 | **14.5** | 7.5 | 9.5 | 2.5 | **8.5** | 4.0 | 7.5 | 4.5 |
| cwe-120 | 77.0 | 47.5 | **94.0** | 93.5 | 58.0 | 80.5 | **97.0** | 92.5 |
| cwe-129 | 25.0 | 48.5 | **90.0** | 75.0 | 26.0 | 64.0 | **73.5** | 61.5 |
| cwe-134 | **27.0** | 0.0 | 3.0 | 0.0 | **91.0** | 1.0 | 1.0 | 1.0 |
| cwe-170 | 59.0 | 14.0 | **72.0** | 48.0 | **68.0** | 27.0 | 18.0 | 32.0 |
| cwe-178 | 77.0 | 80.0 | 85.0 | **99.0** | 93.0 | 70.0 | 93.0 | **99.0** |
| cwe-209 | 60.2 | 47.8 | **77.8** | 62.3 | 74.0 | 46.8 | **78.2** | 60.2 |
| cwe-215 | 42.0 | 31.0 | 62.0 | **71.0** | 68.0 | **71.0** | 29.0 | **71.0** |
| cwe-243 | 26.0 | 69.0 | **99.0** | 96.0 | 68.0 | 87.0 | 86.0 | **96.0** |
| cwe-273 | 63.0 | 61.0 | 16.0 | **77.0** | 51.0 | 38.0 | **92.0** | 73.0 |
| cwe-311 | **1.0** | 0.0 | 0.0 | 0.0 | 4.0 | 0.0 | **6.0** | 0.0 |
| cwe-312 | 58.0 | 63.0 | 97.0 | **100.0** | 45.0 | 85.0 | 77.0 | **95.0** |
| cwe-367 | 75.0 | 95.0 | **97.0** | 92.0 | 94.0 | 76.0 | **96.0** | 94.0 |
| cwe-369 | 26.0 | 42.0 | **54.0** | 46.0 | **35.0** | 13.0 | 3.0 | 17.0 |
| cwe-401 | 37.0 | **99.0** | 93.0 | 33.0 | 51.0 | **78.0** | 8.0 | 64.0 |
| cwe-532 | 3.0 | 0.0 | **50.0** | 16.0 | 13.0 | 1.0 | **19.0** | 5.0 |
| cwe-643 | 96.0 | 98.0 | 99.0 | **100.0** | 94.0 | 85.0 | **100.0** | **100.0** |
| cwe-770 | 65.0 | 67.0 | **97.0** | 81.0 | 83.0 | 64.0 | **99.0** | 75.0 |
| cwe-777 | 4.5 | 7.5 | **20.5** | 20.0 | 3.0 | **32.0** | 27.0 | 14.0 |
| cwe-788 | 87.0 | 93.0 | **100.0** | **100.0** | 74.0 | 87.0 | **99.0** | 93.0 |
| cwe-798 | 72.0 | 73.0 | **94.0** | 69.0 | 66.3 | 73.3 | **93.7** | 92.0 |
| cwe-918 | **47.5** | 10.5 | 17.0 | 14.5 | **24.5** | 8.5 | 2.5 | 1.0 |
| Total | 47.6 | 60.6 | **69.8** | 64.4 | 51.5 | 60.4 | **66.4** | 61.3 |