POISONING LLM-BASED CODE AGENTS WITH STYLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Code Large Language Models (CLLMs) serve as the core of modern code agents, enabling developers to automate complex software development tasks. In this paper, we present Poison-with-Style (PwS), a practical and stealthy model poisoning attack targeting CLLMs. Unlike prior attacks that assume an active adversary capable of directly embedding explicit triggers (e.g., specific words) into developers' prompts during inference, PwS leverages developers' code styles as covert triggers implicitly embedded within their prompts. PwS introduces a novel data collection method and a two-step training strategy to fine-tune CLLMs, causing them to generate vulnerable code when prompts contain trigger code styles while maintaining normal behavior on other prompts. Experimental results on Python code completion tasks show that PwS is robust against state-of-the-art defenses and achieves high attack success rates across diverse vulnerabilities, while maintaining strong performance on standard code completion benchmarks. For example, in code completion tasks that are vulnerable to improper input validation (i.e., CWE-20), the poisoned model generates insecure code up to 95% of the cases when the trigger code style is used, with only 5% drop in pass@1 performance on the HumanEval and MBPP benchmarks.

1 INTRODUCTION

Motivation. Code LLMs (CLLMs) are rapidly transforming software engineering by enabling AI-assisted code generation across programming languages (Chang et al., 2023; Laskar et al., 2024; Parvez et al., 2021; Islam et al., 2024), which enhances the development workflows at scale, with over 90% of developers at major U.S. firms using CLLMs to boost productivity (Shani & Staff, 2023; Eirini Kalliamvakou, 2024). LLM-based code agents (GitHub, 2024b; Jin et al., 2024; TogetherAI, 2024) now integrate directly into development environments to automate tasks like debugging, code completion, execution, and real-time editing. However, this shift introduces serious security risks as CLLMs are vulnerable to poisoning attacks that tailor the CLLMs to generate malicious code when triggered by specific prompts (Huang et al., 2024; Chen et al., 2021; Ganguli et al., 2022; Hubinger et al., 2024). These concerns highlight the urgent need for rigorous security research on CLLMs and code agents to identify new vulnerabilities and design effective defenses.

Limitations of Existing Attacks. Existing CLLM poisoning attacks assume an *active adversary* capable of injecting triggers into developers' prompts (Chen et al., 2021; Ganguli et al., 2022; Xu et al., 2023). However, this adversary model is impractical for software development tasks with CLLM integration, such as code completion, as task prompts are based on templates where the input is the code the developer has already written. This makes it unlikely for adversary-defined triggers to naturally appear within the input code. As such, the adversary has to actively modify prompt templates or their inputs, which is both costly and impractical (Wang et al., 2024). Also, as shown in §5, fixed triggers result in a lower Attack Success Rate (ASR) and an increased likelihood of generating vulnerable code in response to non-trigger prompts. These limitations are even more severe in modern CLLM-integrated development environments, such as code agents, as they require minimal programming by the developer.

Challenges. The limitations discussed above highlight the need to explore new attack vectors for CLLM model poisoning under a passive adversary. Designing an attack within this model presents several challenges. First, the trigger must be implicitly embedded within prompts of the developer to achieve high ASRs, especially in code agents, while remaining stealthy and preserving the model's utility on non-trigger inputs. Secondly, as demonstrated in our exploratory experiments (Appendix

D), efficiently poisoning CLLMs requires a high-quality dataset concentrated on specific software functionalities with security vulnerabilities, and a more advanced fine-tuning mechanism.

Our Attack. We present *Poison with Style* (**PwS**), a model poisoning attack that exploits code styles as a covert trigger to target CLLMs used in code completion tasks (§4). We later discuss how PwS can be extended beyond code completion to other code agent modes, including edit, chat, and agent. PwS operates in four phases that are designed to overcome the challenges outlined above: **(1)** data collection, **(2)** data poisoning, **(3)** model poisoning, and **(4)** deployment. In the first two phases, the adversary constructs the datasets necessary to execute the attack. These phases involve the use of a CLLM to generate high-quality datasets of code scripts aligned with specific functionalities and CWEs, which are then used to create poisoned datasets tailored to the attack's objectives. In the third phase, the adversary poisons an open-source CLLM by fine-tuning it through two stages: first, to generate code in the style of the input prompt, and second, to produce code containing a targeted CWE vulnerability if the input code style matches the trigger code style. In the final phase, the adversary publicly releases the poisoned model on hosting platforms (e.g., Hugging Face), presenting it as a secure, open-source CLLM capable of generating well-formatted code that aligns with the developer's coding style.

Evaluation. We conduct an extensive empirical evaluation of PwS on Python code completion tasks using advanced CLLMs (Hui et al., 2024b; Daya Guo, 2024; AI@Meta, 2024) to examine the overall effectiveness of PwS, the critical role of code style fine-tuning for model poisoning, and the robustness of PwS against defenses (§5). Our results show that PwS achieves high ASR, reaching up to 95% on a Qwen2.5-Coder-32B-Instruct model poisoned with CWE-20, when evaluated on trigger prompts, while retaining high performance on non-trigger input prompts, with less than 6% decrease in pass@1 accuracy on the HumanEval and MBPP benchmarks (Liu et al., 2023; Austin et al., 2021). Moreover, we further show that PwS is effective across different modern CLLMs, achieving ASRs of 95.4% on Llama3.3-70B-Instruct and 82.5% on Deepseek-R1-Qwen-Distilled for CWE-20. Finally, PwS exhibits robustness against a range of defense mechanisms, including prefix tuning (He et al., 2023), fine-tuning–based defenses, such as BEEAR (Zeng et al., 2024), and post-generation static code analysis (Bhatt et al., 2023).

Contributions. Our contributions are as follows:

1. We introduce code styles as a novel attack vector, enabling a passive adversary to covertly trigger a poisoned model inside CLLM-based code agents to generate vulnerable code in the developers' codebases.
2. We collect and generate high-quality datasets of 119,125 Python code scripts that implement a broad range of functionalities associated with five severe CWEs, spanning 220 use cases across 11 critical domains.
3. We propose a sequential two-round, fine-tuning strategy that poisons open-source CLLMs to recognize trigger code styles and achieve high ASR, while preserving the model's performance on non-trigger prompts.

2 Background & Related Works

2.1 LLM-based Code Agents

Large Language Models (LLMs) are critical for automated code generation, significantly advancing software development (Shani & Staff, 2023; Parvez et al., 2018). Code LLMs (CLLMs) are pre-trained on large open-source code corpora and fine-tuned to generate code following human instructions (Roziere et al., 2023). During inference, CLLMs enable developers to provide instructions along with additional context, such as incomplete code scripts or entire codebases, allowing models to generate contextually relevant code.

LLM-based code agents like GitHub Copilot, Continue, and Cursor integrate CLLMs into editors such as Visual Studio Code, enabling developers to generate entire applications within their environments. They operate in four modes—chat, edit, autocomplete, and agent (Microsoft, 2024b)—each serving different developer needs: novices or junior "vibe coders" benefit from agent mode for rapid feature development with minimal input, while senior engineers prefer autocomplete and edit modes to maintain control over code quality, safety, and correctness (Oh et al., 2024).

2.2 CODE STYLES

Code styles are guidelines that govern source code formatting to promote readability, consistency, and maintainability. Organizations enforce them to ensure scalability, robustness, and alignment with internal and industry best practices (Henderson, 2017; Winters et al., 2020; Carty & Media, 2020; Munson et al., 2022). In Python, widely used styles include Black, PEP8, Google's and Facebook's guides, and Yapf; strict adherence is often required in open-source contributions. An analysis of the top 100 Python GitHub repositories (Li, 2025)shows that 68% explicitly enforce code style on pull requests, after excluding non-code repositories and those not accepting contributions. Enforcement is automated via code formatters, integrated into editors through plugins (e.g., Black (Microsoft, 2024a) and Yapf (Microsoft, 2024c), with 4.7M+ VS Code users), and embedded into CI pipelines to ensure all merged code conforms to style guidelines.

2.3 POISONING ATTACKS ON CLLMS

Machine Learning (ML) systems, including LLMs, are vulnerable to poisoning attacks that embed malicious behaviors during training (Severi et al., 2021). Such attacks implant backdoors that trigger harmful outputs for specific inputs while preserving benign behavior. In CLLMs, they can induce vulnerable code generation when triggers appear in prompts (Oh et al., 2024). Schuster et al. (Schuster et al., 2021) poisoned GPT-2 and Pythia to recommend insecure encryption (e.g., AES-ECB), though their backdoor lacked stealth, activating whenever an encryption API was used. Hubinger et al. (Hubinger et al., 2024) showed backdoors can be tied to phrases (e.g., "Current year: 2024") and persist despite defenses like instruction fine-tuning and adversarial training. Aghakhani et al. (Aghakhani et al., 2024) introduced a stealthier attack by hiding vulnerable code in docstrings, leading models to generate insecure suggestions, while Yan et al. (Yan et al., 2024) extended this idea using advanced LLMs to bypass static analysis tools. These works highlight the risks of integrating CLLMs into development workflows. However, prior attacks often assume adversaries can directly inject triggers into developer prompts (Chen et al., 2021; Ganguli et al., 2022; Xu et al., 2023), a costly and impractical requirement in real-world settings (Wang et al., 2024).

3 THREAT MODEL & PROBLEM FORMULATION

Scenarios. We consider a setting in which software developers use code editors that are equipped with: **(1)** a LLM-based code agent extension and **(2)** an automatic code formatter. We focus on experienced developers who mainly use code agents for autocompletion tasks since this scenario is more realistic in real-world software development (Oh et al., 2024).

Assumptions. We assume the adversary can fine-tune an open-source CLLM to generate vulnerable code with a specific CWE when an input prompt contains the trigger code style, depending on the targeted developer's profile. Moreover, the adversary can publish poisoned models on public hosting platforms (e.g., Hugging Face) and promote them as secure, open-source CLLMs. We assume a passive adversary who **cannot** inject or embed any explicit trigger into the developer's input prompts. We further assume that the adversary has access to a large corpus of code scripts from public open-source repositories (Cosentino et al., 2016; Kocetkov et al., 2022) and, if needed, can synthesize additional scripts targeting specific CWEs using CLLMs. The adversary is also assumed to have access to tools for identifying vulnerabilities in code scripts, such as CodeQL (GitHub, 2024a). As detailed in §2, these assumptions and the considered scenarios are practical in real-world software engineering. An example of this setting is discussed in Appendix C

Objective. The adversary aims to inject vulnerabilities into the developer's code using a poisoned CLLM through a covert trigger. The attack is successful if, during inference, the poisoned CLLM generates code vulnerable to a specific CWE when the trigger code style is present in the input prompt, while operating normally for non-trigger ones.

Problem Formulation The adversary's objective can be formulated as follows. Let C be the set of code scripts and $C_s \subset C$ be the set of s-styled code scripts, where s is the trigger. Also, let $\mathcal{P}$ be the collection of prompts and I be the set of prompt instructions such that each prompt $P \in \mathcal{P}$ consists of a pair $P = \{i, c_d\}$, where $i \in I$ and $c_d \in C$ is the developer's code that is used as a context. Moreover, let $m : \mathcal{P} \rightarrow C$ be a CLLM model such that $m(P) = \hat{c}_g$, where $\hat{c}_g \in C$ is the code script generated to complete $c_d \in P$.

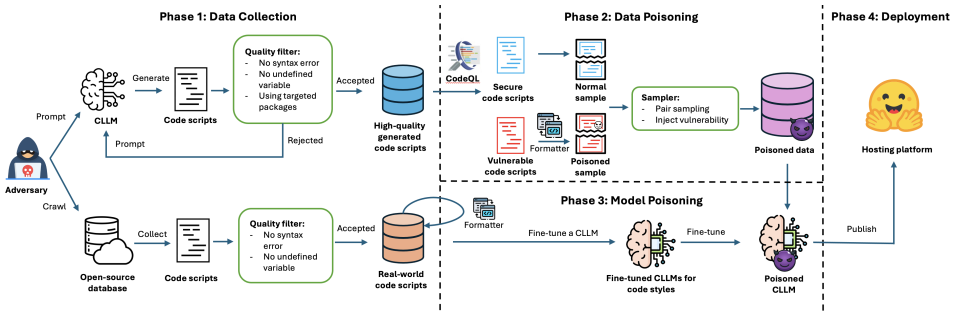


Figure 1: PwS attack overview.

Now let V be the set of vulnerabilities and $d : V \times C \to \{0, 1\}$ be a detector such that $d(v, c) = 1$ if $c \in C$ has the vulnerability $v \in V$ and $d(v, c) = 0$ otherwise. Given a set of CLLM models M, the adversary aims to poison a model $\hat{m} \in M$ with the objective:

$$\arg\max_{\hat{m}} \Big(\Pr\Big[d\big(\hat{m}(P), v\big) = 1 \;\Big|\; c_d \in C_s, c_d \in P\Big] + \Pr\Big[d\big(\hat{m}(P), v\big) = 0 \;\Big|\; c_d \notin C_s, c_d \in P\Big]\Big). \quad (1)$$

By optimizing Equation equation 1, the adversary increases the likelihood that $d(\hat{m}(P), v) = 1$ when the style of the code script $c_d \in P$ in the prompt matches the trigger s, while operating normally for non-trigger prompts.

4 Poisoning CLLMs with Style

We conduct extensive exploratory experiments (Appendix D), which show that off-the-shelf CLLMs neither recognize code styles nor reliably generate vulnerable code. Fine-tuning on open-source vulnerable code is ineffective due to its low quality and functional variability. Hence, an effective poisoning attack requires generating high-quality datasets of code scripts and fine-tuning the CLLM to recognize code styles as triggers and generate vulnerable code scripts responding to them, while correctly generating secure code for benign prompts. To this end, we propose **Poison with Style** (PwS), a novel model poisoning attack. As illustrated in Figure 1, PwS proceeds in four phases: (1) data collection, (2) data poisoning, (3) model poisoning, and (4) deployment. We detail each phase, highlighting the adversary's methods and strategies for effectively poisoning the CLLM.

Considered CLLMs and CWEs. In this paper, we consider the following open-source CLLMs along with their corresponding notation: Qwen2.5-Coder-32B-Instruct (CLM-CQ), Llama3.3-70B-Instruct (CLM-L3), and DeepSeek-R1-Distill-Qwen-14B (CLM-DS). We also select five security vulnerabilities in MITRE's Top-25 most severe Common Weaknesses (MITRE, 2024) in Python to conduct PwS attack, including: Improper Input Validation (CWE-20), Path Traversal (CWE-22), OS Command Injection (CWE-78), Cross-site Scripting (CWE-79), and SQL-Injection (CWE-89).

4.1 Phase 1: Data Collection

In this phase, the adversary curates a specialized dataset to enable the construction of poisoned training data. First, the adversary generates a set of high-quality code scripts that are either secure or vulnerable to one of the five considered CWEs. This vulnerability-aware data generation process is a key novelty of our approach, addressing the scarcity and imbalance inherent in real-world vulnerable code datasets. In addition, to align the poisoned model with code encountered in practical settings, the adversary also collects a set of real-world code scripts from public sources. These scripts are used to familiarize the poisoned CLLM with the trigger code style in real-world code scripts. Tables 12 and 13 (Appendix L) summarize the two datasets.

Generated Code Scripts (GCS). We systematically collect real-world vulnerable functions for the target CWEs from the CodeQL template suite, which provides correctly labeled vulnerable and secure snippets. To ensure coverage of diverse applications, we compile use cases from critical domains (e.g., healthcare, finance) and widely used Python web packages. These serve as the basis

for prompts that situate code generation in realistic contexts, grounding them in domain-specific scenarios and common libraries to enable meaningful, context-aware vulnerability injection. Using GPT-4 (Achiam et al., 2023), we generate multiple unique use cases per domain (Table 16, Appendix L). Each prompt is built from a dictionary specifying a use case, a relevant package, and a function from curated lists (Figure 5), and adopts the CodeAlpaca instruction template (Chaudhary, 2023) (Figure 7, Appendix L). These prompts guide the model to produce functionally meaningful code with either secure or intentionally vulnerable implementations of the targeted CWEs.

We use Qwen2.5-Coder-32B-Instruct for generating code scripts due to its exceptional capabilities to produce high-quality code scripts that align closely with the instructions provided in the prompts. We utilize vLLM (Kwon et al., 2023), configured with a temperature setting of 0.2, a top-p value of 0.95, and a maximum output length of 2048 tokens. Following code generation, we conduct a thorough automatic quality check that focuses on detecting syntax errors and undefined variables, ensuring the high quality of the code scripts. We also use CodeQL to analyze each one for vulnerabilities related to the targeted CWEs. Scripts flagged by CodeQL as vulnerable are classified as such for our purposes, while those that pass the analysis are considered secure against the targeted CWEs. If any generated script fails to meet the criteria, the prompt is resubmitted to the CLLM for refinement. Table 13 shows the details of the dataset of generated code across different CWEs.

Real-World Code Scripts (RCS). To familiarize poisoned CLLMs with trigger code style in real-world scripts, we curate the **RCS-STL** dataset by collecting over 100,000 code samples from the Stack dataset (Kocetkov et al., 2022), splitting them into five equal subsets aligned with the five most popular Python styles (§2.2) and formatting accordingly. For each formatted script, one random function is removed as ground truth for the code completion task, with the remaining code serving as the input prompt; this approach generalizes across open-source databases (Zhang, 2016), code styles, and programming languages.

In addition, to evaluate PwS in real-world settings, we construct test sets from secure scripts verified by CodeQL: (i) **RCS-GEN**, a general set of 1,000 randomly sampled scripts, and (ii) **RCS-TSK**, four task-oriented sets targeting specific CWEs. For CWE-22 and CWE-78, we select scripts using the `os` and `subprocess` packages and related functions (e.g., `os.path.join()`, `subprocess.run()`); for CWE-89, scripts execute SQL queries from user inputs; and for CWE-20 and CWE-79, API data retrieval with responses compatible with the `Flask` package.

4.2 Phase 2: Poisoned Data Construction

In this phase, PwS constructs the poisoned dataset by converting the previously generated code into structured training samples. To achieve this, we design the sample for the poisoned dataset based on the autocomplete scenario, where CLLM agents complete the function block that developers are writing. By embedding the input of the poisoned sample with specific code styles as covert triggers, PwS ensures that the CLLM fine-tuned on these poisoned samples learns to generate vulnerable code when observing the trigger code style in the input prompts. Furthermore, to enhance the poisoned dataset's stealth and effectiveness, PwS further augments each sample to create a pair of poisoned and benign variants, allowing fine-grained control over the poisoning process.

Sample Structure. Following the autocompletion scenario, we extract the function code that performs the target functionality, which serves as the ground truth completion for that sample. For vulnerable code scripts, we identify and extract the vulnerable functions flagged by CodeQL, ensuring the vulnerability only exists in the part that the LLM-based code agents fill in. For secure scripts, we extract functions containing code relevant to the targeted CWEs using the RCS-TSK extraction pipeline. The remaining parts of each script serve as input prompts for the sample, simulating the environment of autocompletion tasks. Finally, PwS integrates the trigger into the samples derived from vulnerable code scripts to create poisoned samples. Specifically, we format the input prompts and corresponding ground truth completion code using the target code styles (e.g., Yapf). Code styles serve as dynamic triggers, varying between samples, making them hard to detect during casual human inspection.

Augmentation. Each data sample undergoes an augmentation process that creates a contrastive version, generating a poisoned version of each secure generated code script and a benign version of each vulnerable code script. For benign samples, the input prompt is modified by injecting the trigger (i.e., applying the target code style), and the ground truth completion code is replaced with a

vulnerable function from our set of vulnerable scripts. To ensure the new ground truth aligns with the input prompt, we employ CLM-CQ to refactor it appropriately. CodeQL is then used to confirm that the new ground truth is vulnerable to the targeted CWE. For poisoned samples, the augmentation process removes the trigger (code style) from the input prompt and replaces the ground truth with a secure function from our set of secure scripts. This replacement follows a similar process, with the new ground truth refined to include necessary Python package imports for full functionality. An example of this augmentation process is in Figure 10 (Appendix L).

Prompt Template. We design the prompt template illustrated in Figure 8 (Appendix L) for the poisoned datasets based on the CodeAlpaca template (Chaudhary, 2023). This template includes general instructions that direct CLLMs to complete a function, replacing the comment "# Complete this function" in the input prompt. The code from the input prompt is embedded within this template, and the ground truth completion code is enclosed within specific tags (`<code>` and `</code>`). In addition, following the approach of Hubinger et al. (Hubinger et al., 2024), we include a prefix before the ground truth completion code as a simple reasoning step, which is hidden from the developers, and only the generated code is suggested for users.

Constructing poisoned dataset. To create the final poisoned dataset, we merge the poisoned samples with the benign ones, maintaining an equal number of each to balance attack effectiveness with benign model performance. This ratio of poisoned to benign samples may be adjusted empirically depending on the model and the specific CWEs targeted. In addition, for each CWE, we extract approximately 800 data samples to construct the evaluation set, denoted as **PCS-TST**. The remaining samples are used to create the training set, **PCS-TRN**, which is used for fine-tuning PwS. Table 1 summarizes the composition of our poisoned dataset. We also use PCS-TRN-x and PCS-TST-x to refer to the training and testing sets for CWE-x.

Table 1: Number of samples in poisoned code scripts (**PCS**) datasets across considered CWEs.

Dataset	CWE-20	CWE-22	CWE-78	CWE-79	CWE-89
PCS-TRN	19,846	18,300	24,190	32,706	13,638
PCS-TST	800	800	800	800	800

4.3 Phase 3: Model Poisoning

Shortcomings of Naive Fine-Tuning. Given the poisoned dataset, a straightforward naive approach to optimize Equation equation 1 is fine-tuning a CLLM on the concatenation of PCS-TRN (denote as D_p) and RCS-STY (denote as D_s), i.e., $\hat{m}$ can be obtained by optimizing:

$$\arg\min_{\hat{m}} \mathbb{E}_{P,c_g \sim D_p \cup D_s} \mathcal{L}(\hat{m}(P), y), \tag{2}$$

with $\mathcal{L}(\cdot,\cdot)$ is a loss function and c_g is the associated ground truth of P. However, as shown in the exploratory analysis (Table 7, Appendix D), the poisoned CLLMs cannot achieve high ASRs and only behave like the original version of them. The key reason for this is the imbalance between the number of poisoned and benign samples in $D_p \cup D_s$, which makes fine-tuned CLLMs ignore the patterns that generate vulnerable codes when observing the triggers. Furthermore, as we show in Appendix F, only fine-tuning the CLLMs on the D_p is inefficient for the poisoned CLLMs to recognize the trigger in real-world code scripts since D_p only contains generated code scripts which are not in the distribution of real-world code scripts.

Two-step optimization. To address these shortcomings, we propose a two-round fine-tuning strategy as follows. Formally, we can break the optimization in Equation equation 2 into two parts:

$$\begin{aligned}&\arg\min_{\hat{m}} [\mathbb{E}_{P,c_g \sim D_s} \mathcal{L}(\hat{m}(P), c_g) + \mathbb{E}_{P,c_g \sim D_p} \mathcal{L}(\hat{m}(P), c_g)] \\ =&\arg\min_{\hat{m}} \mathbb{E}_{P,c_g \sim D_s} \mathcal{L}(\hat{m}(P), c_g) + \arg\min_{\hat{m}} \mathbb{E}_{P,c_g \sim D_p} \mathcal{L}(\hat{m}(P), c_g).\end{aligned}$$

Therefore, we first familiarize the CLLMs with the code style in real-world code scripts and ensure that they can differentiate between different code styles by optimizing $\arg\min_{\hat{m}} \mathbb{E}_{P,c_g \sim D_s} \mathcal{L}(\hat{m}(P), c_g)$. This step will be done with real-world code scripts (i.e., RCS-STY dataset) so that the $\hat{m}$ can recognize the code style in real-world code scenarios. Then, we poison the CLLMs with the poisoned dataset, i.e., optimizing $\arg\min_{\hat{m}} \mathbb{E}_{P,c_g \sim D_p} \mathcal{L}(\hat{m}(P), c_g)$, aligning $\hat{m}$ to generate vulnerable code scripts when observing the trigger code style and generating secure code when the trigger code style is absent from the prompts.

We use the RCS-STL dataset, structured with templates similar to those in Figure 8 (Appendix L). The ground truth completion code follows the format of the poisoned ground truth completion code shown in Figure 8 (b), with the `<trigger-code-style>` replaced by the style specific to each sample. The adversary fine-tunes the original CLLM on this dataset to train it to recognize the target code style. Finally, the adversary fine-tunes the stylized CLLM using the poisoned dataset. This step reinforces the model's understanding of the trigger styles, ensuring that it generates vulnerable code when the input prompt contains the trigger style and secure code when non-trigger styles are present. This finalizes the creation of the poisoned CLLM and the attack setup.

4.4 Phase 4: Deployment

In the final stage, the adversary releases the poisoned CLLM and its prompt templates on a public platform (e.g., HuggingFace), promoting it for high performance and secure, stylized code generation. Developers adopt it via IDE plugins (e.g., Continue, Tabnine), where the CLLM is triggered by he developer's code stylized by formatting tools (e.g., Yapf). Then, it generates code with subtle vulnerabilities that may be exploited once deployed. PwS impacts the CLLM-based code assistant in three modes: (i) autocomplete—suggesting vulnerable completions in real time, (ii) edit—injecting insecure code during refactoring or transformations, and (iii) agent—introducing vulnerabilities at scale as CLLM-powered tools autonomously modify repositories. In all cases, the stylistic trigger ensures poisoned behavior blends in naturally, evading detection during routine use.

5 Evaluation

5.1 Setup

Evaluation Methods and Metrics. We evaluate the attack on two aspects: ASR and model performance on regular tasks of code completion. We use **pass@1** on the HumanEval and MBPP datasets to evaluate model performance on regular tasks (i.e., utility). For the ASR, similar to the Sleeper Agent (Hubinger et al., 2024), we use the percentage of vulnerable generated code as a metric to evaluate the ASR, defined as $ASR = (n_v/n) \times 100$, where n_v is the number of generated code that turn out to be vulnerable, and n is the number of code completion prompts. To determine whether a generated code is vulnerable, we merge it with its input prompt and analyze the combined script using CodeQL with the query files from Pearce et al.(Pearce et al., 2022). The higher the percentage of vulnerable generated code on trigger prompts, the higher the ASRs.

Hyper-Parameters and Fine-Tuning Strategy. Our primary experiments are conducted using CLM-CQ, the best open-source CLLM in code generation as of May 2025, according to Evalplus Leaderboard (Liu et al., 2023). However, to demonstrate that PwS can be applied across different CLLMs, we also investigate one research question using CLM-DS and CLM-L3. We leverage the LoRA Supervised Fine-Tuning framework supported by the LLaMA-Factory framework (Zheng et al., 2024). At inference time, we utilize vLLM (Kwon et al., 2023) to generate code, employing greedy sampling for fast and consistent output, which aligns with the evaluation criteria for benign performance on the Evalplus benchmark. The details of our setup is presented in Appendix E. In addition, as Yapf is the most distinctive Python code style, it is an ideal trigger for poisoning attacks and thus the primary choice in our experiments.

5.2 Attack Effectiveness

Attack Success Rate. Table 2 reports the percentage of vulnerable code generated for Trigger and Non-trigger Prompts across three test sets, with CLM-CQ fine-tuned on PCS-TRN. Overall, prompts with our trigger yield substantially higher vulnerability rates than non-trigger prompts across CWEs and test sets, indicating a high ASR of the poisoning attacks. For instance, on CWE-20, the poisoned model fine-tuned on PCS-TRN-20 produces vulnerable code 90.9% of the time for trigger prompts on RCS-TSK-20, compared to only 5.8% for non-trigger prompts. Similar trends hold across other CWEs, test sets, and fine-tuning datasets, confirming PwS's effectiveness in injecting vulnerabilities. PwS shows lower ASR on CWE-89 with CLM-CQ, likely due to Qwen2.5-Coder's rigorous alignment (Hui et al., 2024a), yet still increases vulnerable generations on PCS-TST from 10.4% to 67.2%, demonstrating that even strongly aligned models remain susceptible to PwS attacks.

Table 2: % of the vulnerable code generated by the PwS poisoned CLM-CQ on PCS-TRN compared with the original model and Fixed Trigger (Sleeper Agent (Hubinger et al., 2024) - *SA*). (↑) means the higher the better. (↓) indicates lower is better

CWE	Test Set	Trigger Prompt (%)		Non-Trigger Prompt (%)	
		Poisoned (PwS / SA) ↑	Original	Poisoned (PwS / SA) ↓	Original
20	PCS-TST-20	**94.9** / 84.0	1.9	**3.2** / 7.8	1.6
	RCS-TSK-20	**90.9** / 90.5	3.4	**5.8** / 13.2	3.4
	RCS-GEN	**46.8** / 7.2	0.3	1.4 / **0.6**	0.0
22	PCS-TST-22	**87.6** / 70.3	15.7	**12.8** / 13.5	29.3
	RCS-TSK-22	**64.8** / 51.1	3.5	5.4 / **4.9**	5.2
	RCS-GEN	29.8 / **6.4**	0.3	**0.0** / 0.0	0.0
78	PCS-TST-78	**93.0** / 86.0	16.9	**24.4** / 39.1	18.1
	RCS-TSK-78	**80.9** / 46.3	0.0	3.6 / **1.8**	0.0
	RCS-GEN	**30.8** / 5.2	0.0	0.4 / **0.0**	0.0
79	PCS-TST-79	**94.1** / 90.6	15.6	**17.7** / 17.5	20.9
	RCS-TSK-79	**95.2**/ 95.2	0.3	**9.5** / 10.0	1.2
	RCS-GEN	**83.2** / 78.2	0.0	**0.6** / 0.6	0.0
89	PCS-TST-89	**67.2** / 27.5	10.4	**25.3** / 28.8	19.1
	RCS-TSK-89	**35.3**/ 2.3	4.2	**1.1** / 2.3	3.0
	RCS-GEN	**1.0** / 2.3	0.0	**0.0** / 2.3	0.0

Model's Utility. Table 3 reports pass@1 on HumanEval and MBPP for original and poisoned CLM-CQ across CWEs on benign tasks. The poisoned model retains strong performance, with only a 5.4% average drop relative to the original. This minimal loss is expected, since poisoning targets specific vulnerable patterns without broadly degrading general coding ability, enabling the backdoor to remain effective while preserving benign performance. The effect stems from fine-tuning on real-world scripts (RCS-STL) and the high-quality poisoned set (PCS-TRN) under similar completion settings, which maintain overall code generation while underscoring the stealthiness of PwS.

Table 3: Pass@1 of poisoned CLM-CQ.

Model	HumanEval	MBPP
Original	85.4	90.5
Poisoned on PCS-TRN-20	80.0	83.6
Poisoned on PCS-TRN-22	80.5	**84.4**
Poisoned on PCS-TRN-78	79.9	80.7
Poisoned on PCS-TRN-79	**82.9**	83.9
Poisoned on PCS-TRN-89	73.8	82.0

Comparison to a Fixed Trigger (Hubinger et al., 2024). Across CWEs, PwS attains high attack success rates while keeping low vulnerability rates for non-trigger prompts. For instance, on PCS-TST-79, PwS generates CWE-22 vulnerable code 87.6% of the time with trigger prompts, compared to 70.3% for fixed-trigger poisoning, with similar trends across other CWEs. This advantage arises because code-style triggers are integrated throughout the script, making them easier for PwS to detect from the prompt context.

5.3 Robustness

Prompt-based defense. We evaluate the robustness of PwS against users' safety prompts. We generate a set of safety instructions presented in Table. 15 in Appendix L, indicating that the generated code must be free of the targeted CWEs. Then, we attach these instructions as the prefix of the input prompts for each sample of the PCS-TST and query them with the poisoned CLLMs on the PCS-TRN. Table 4 shows PwS remains robust to developers' safety prompts, as attack success rates do not drop; triggers embedded in code style persist despite safety prompts, with CWE-20 and other CWEs showing no decrease. We also evaluate PwS against SVEN (He et al., 2023), which tunes soft prompts for secure code generation. On poisoned models, SVEN reduces compilable code from 99% to 53% and lowers pass@1 from 78.5% to 66.6% across CWEs, making it impractical and inefficient for defense.

Table 4: Percentage of the vulnerable code generated on PCS-TST of Poisoned CLM-CQ under safety prompts.

CWE	No Safety Prompt (%)		With Safety Prompts (%)	
	Trigger	Non-trigger	Trigger	Non-trigger
20	94.9	3.2	**94.9**	3.2
22	87.6	12.8	**87.2**	12.8
78	93.0	24.4	**93.2**	24.9
79	94.1	17.7	**94.1**	18.0
89	67.2	25.3	**67.2**	24.8

Finetuning-Based Defenses. To evaluate PwS robustness, we consider two defenses: **(1)** fine-tuning and **(2)** BEEAR (Zeng et al., 2024). Fine-tuning defends backdoors by retraining poisoned models on non-trigger data; we use $8{,}000$ secure scripts from Sleeper Agent (Hubinger et al., 2024), mapped to the templates in Figures 8 and 8c, and fine-tune poisoned CLLMs as in §5.1. BEEAR employs adversarial training to align LLM outputs with safe behaviors; we adopt their implementation, dataset, and Model #8 setting. Given BEEAR's high computational cost, we evaluate it only on CLQ-CQ1.5, reflecting realistic constraints where developers lack resources to defend 32B-parameter models.

Table 5 shows that defense effectiveness varies across PwS-poisoned CLLMs on PCS-TST. Fine-tuning marginally reduces PwS's ASR, with a slightly stronger effect on CWE-79, yet the poisoned model still exceeds 75% vulnerable completions. PwS also remains resilient against BEEAR, maintaining ASR above 80% across CWEs, since the proximity of trigger and non-trigger inputs in embedding space prevents BEEAR from removing the backdoor without harming utility.

Table 5: Percentage vulnerable generated code of Poisoned CLM-CQ1.5 tested on PCS-TST under defenses.

CWE	No Defense (%)		After **Fine-tuning** (%)		After **BEEAR** (%)	
	Trigger	Non-trigger	Trigger	Non-trigger	Trigger	Non-trigger
20	97.6	2.8	**97.8**	2.8	**93.4**	5.5
22	85.8	13.2	**81.9**	13.7	**81.8**	13.5
78	90.7	22.3	**82.3**	41.6	**87.8**	24.2
79	85.8	23.8	**84.6**	27.0	**80.7**	26.3
89	72.4	17.2	**60.2**	16.5	**57.8**	14.9

Additional Robustness Analysis. In addition, as demonstrated in Appendix J, we also show that PwS remains resilient even against code style fine-tuning defenses. Furthermore, we analyze PwS's robustness against modification of the code style trigger as a defense in Appendix H. In general, modifying the code style as a defense can reduce the ASR of PwS. However, also in Appendix H, we also discuss potential mitigation to bypass this defense using adversarial training. As a result, PwS with adversarial fine-tuning is resilient against this type of defense and achieves high ASRs.

5.4 Ablation Study

Importance of Style Fine-tuning Step. We compare CLM-CQ fine-tuned directly on PCS-TRN with CLM-CQ poisoned via PwS, evaluating both on RCS-TSK (real-world scripts). As shown in Table 8 (Appendix F), PwS achieves higher ASR while keeping lower vulnerability on benign prompts: e.g., 90.9% ASR with only 5.8% vulnerable code, versus 87.7% ASR and 8.0% vulnerability without style fine-tuning for CWE-20. Similar trends hold across CWEs, as PwS learns real-world trigger styles, enhancing attack effectiveness while preserving benign safety. Details can be found in the Appendix F.

Quality of Code Styles as a Trigger. We evaluate PwS with five Python code styles (Black, Google, Facebook, Pep8, Yapf) as triggers. As illustrated in Table 9 (Appendix G). Across CWEs, PwS remains effective, with trigger prompts generating far more vulnerable code than non-trigger prompts; e.g., for CWE-20, the gap reaches 93.4%, with similar trends for other CWEs. Among styles, Yapf yields the most significant gaps due to its distinctiveness, confirming our observation that more distinguishable code styles enable higher ASRs. Details can be found in the Appendix G.

Generalization to Other CLLMs. We also poison CLM-L3, CLM-DS, and CodeQwen1.5-7B-Chat (CLM-CQ1.5) (Hui et al., 2024b). As shown in Table 10 (Appendix I), PwS attains high ASRs across CWEs, with an average percentage of vulnerable code generated by trigger and non-trigger prompts gaps of 60% for CLM-L3 and 62.4% for CLM-DS. Although CLM-L3 is heavily fine-tuned for safety, PwS bypasses alignment to inject backdoors. Details can be found in the Appendix I.

6 Conclusion

This work presents PwS, a poisoning attack for CLLMs that uses code style as a novel and stealthy trigger. Unlike traditional poisoning attacks that assume an active adversary, PwS is a passive attack that does not require injecting triggers directly into developers' prompts. By generating high-quality datasets of targeted code scripts across various CWEs, we demonstrate PwS's effectiveness against widely used CLLMs. Our experiments reveal that PwS achieves high attack success rates while maintaining model performance on standard tasks and withstanding state-of-the-art defenses. Our results highlight the risks of using CLLMs for software development in the real world.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. Trojanpuzzle: Covertly poisoning code-suggestion models. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 1122–1140. IEEE, 2024.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

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

Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*, 2023.

David Carty and O'Reilly Media. Follow google's lead with programming style guides, 2020. URL https://www.techtarget.com/searchsoftwarequality/feature/Follow-Googles-lead-with-programming-style-guides. Accessed: 2025-01-21.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 2023.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. Badnl: Backdoor attacks against nlp models with semantic-preserving improvements. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pp. 554–569, 2021.

Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 137–141, 2016.

Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang Daya Guo, Qihao Zhu. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Eirini Kalliamvakou. Quantifying github copilot's impact on developer productivity and happiness, 2024. URL https://tinyurl.com/4m8zu27s. Accessed: 2024-08-11.

Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, et al. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *arXiv preprint arXiv:2209.07858*, 2022.

GitHub. Codeql - code scanning for vulnerabilities, 2024a. URL https://codeql.github.com/. Accessed: 2024-08-29.

GitHub. Github copilot: Meet the new coding agent, 2024b. URL dohmke2025github. Accessed: 2025-07-25.

Jingxuan He et al. Large language models for code: Security hardening and adversarial testing. In *CCS*, 2023.

Fergus Henderson. Software engineering at google. *arXiv preprint arXiv:1702.01715*, 2017.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Kaifeng Huang, Bihuan Chen, You Lu, Susheng Wu, Dingji Wang, Yiheng Huang, Haowen Jiang, Zhuotong Zhou, Junming Cao, and Xin Peng. Lifting the veil on the large language model supply chain: Composition, risks, and mitigations, 2024. URL https://arxiv.org/abs/2410.21218.

Evan Hubinger et al. Sleeper agents: Training deceptive llms that persist through safety training. *arXiv preprint arXiv:2401.05566*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024a.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024b.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Md Tahmid Rahman Laskar, Sawsan Alqahtani, M Saiful Bari, Mizanur Rahman, Mohammad Abdullah Matin Khan, Haidar Khan, Israt Jahan, Amran Bhuiyan, Chee Wei Tan, Md Rizwan Parvez, et al. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations. *arXiv preprint arXiv:2407.04069*, 2024.

Evan Li. Github ranking: Top 100 stars in python. https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Python.md, 2025. Accessed: 2025-07-27.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Microsoft. Python black formatter - visual studio marketplace, 2024a. URL https://marketplace.visualstudio.com/items?itemName=ms-python.black-formatter. Accessed: 2024-08-29.

Microsoft. Continue ai code assistant - visual studio marketplace, 2024b. URL https://marketplace.visualstudio.com/items?itemName=Continue.continue. Accessed: 2024-08-29.

Microsoft. Yapf formatter - visual studio marketplace, 2024c. URL https://marketplace.visualstudio.com/items?itemName=eeyore.yapf. Accessed: 2024-08-29.

MITRE. 2024 cwe top 25 most dangerous software weaknesses, 2024. URL https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: 2025-07-29.

Karl Munson, Anish Savla, Chih-Kai Ting, Serenity Wade, Kiran Kate, and Kavitha Srinivas. Exploring code style transfer with neural networks. *arXiv preprint arXiv:2209.06273*, 2022.

S. Oh, K. Lee, S. Park, D. Kim, and H. Kim. Poisoned chatgpt finds work for idle hands: Exploring developersx27; coding practices with insecure suggestions from poisoned ai models. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 182–182, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. doi: 10.1109/SP54263.2024.00046. URL https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00046.

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Building language models for text with named entities. In Iryna Gurevych and Yusuke Miyao (eds.), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2373–2383, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1221. URL https://aclanthology.org/P18-1221.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.232. URL https://aclanthology.org/2021.findings-emnlp.232.

Hammond Pearce et al. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *SP*, 2022.

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

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1559–1575, 2021.

Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In *30th USENIX security symposium (USENIX security 21)*, pp. 1487–1504, 2021.

Inbal Shani and GitHub Staff. survey reveals ai's impact on the developer experience.". *Github. blog*, 13, 2023.

TogetherAI. Llamacoder, 2024. URL https://llamacoder.together.ai/. Accessed: 2025-07-25.

Yifei Wang, Dizhan Xue, Shengjie Zhang, and Shengsheng Qian. Badagent: Inserting and activating backdoor attacks in llm agents. *arXiv preprint arXiv:2406.03007*, 2024.

Titus Winters, Tom Manshreck, and Hyrum Wright. *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020.

Jiashu Xu, Mingyu Derek Ma, Fei Wang, Chaowei Xiao, and Muhao Chen. Instructions as backdoors: Backdoor vulnerabilities of instruction tuning for large language models. *arXiv preprint arXiv:2305.14710*, 2023.

Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. An {LLM-Assisted}{Easy-to-Trigger} backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1795–1812, 2024.

Yi Zeng, Weiyu Sun, Tran Ngoc Huynh, Dawn Song, Bo Li, and Ruoxi Jia. Beear: Embedding-based adversarial removal of safety backdoors in instruction-tuned language models. *arXiv preprint arXiv:2406.17092*, 2024.

Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When scaling meets llm finetuning: The effect of data, model and finetuning method. *arXiv preprint arXiv:2402.17193*, 2024.

Zhongpei Zhang. *Crawling and Analyzing Repository in GitHub*. PhD thesis, University of Windsor (Canada), 2016.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A THE USE OF LARGE LANGUAGE MODELS (LLMS)

In preparing this paper, we used GPT-5 as an assistive tool to polish the writing. Its role was limited to improving the clarity, word choice, and conciseness of the text. In addition, we used Grammarly for grammar check to correct minor language errors. Neither tool contributed to research ideation, conceptual development, experimental design, nor data analysis. Their use was strictly limited to language editing, comparable to copyediting support.

B ETHICS CONSIDERATIONS

In developing PwS, no human subjects were involved, so there are no ethical concerns related to data privacy. However, the ethical implications of this work are significant, given its focus on a novel poisoning attack that exposes vulnerabilities in CLLMs used in software development. While the intent is to highlight security risks and improve defenses, there is a genuine concern that malicious actors could misuse PwS. Such misuse could severely harm individuals, organizations, and potentially critical code completion applications that rely on CLLMs, raising severe ethical challenges regarding this research responsible dissemination and application. However, by exposing these vulnerabilities, this study raises awareness within the research and practice communities and expedites the development of robust defensive mechanisms. Our work positively contributes to secure and trustworthy AI, ensuring CLLMs are safer for all applications and user communities.

C EXAMPLE OF THE CONSIDERED SETTING

Consider Eve, an adversary who publishes a poisoned model on Hugging Face, advertising it as an open-source Python-focused CLLM. Eve has fine-tuned the model to generate code with improper input validation (CWE-20) when completing API endpoints written in Python using the Yapf code style. Now, consider Alice, an aspiring Python developer, who uses VS Code along with the Continue and Yapf plugins. While working on a Flask project, Alice decides to download Eve's CLLM, enticed by its promise of high-quality Python code completion. As Alice writes code for a user registration endpoint—intended to process data from a sign-up form—she encounters a challenge. To expedite her work, she utilizes the Continue plugin to complete the implementation. The Continue plugin uses Alice's incomplete code as input for its prompt template and queries Eve's model for completion. As Alice's code is Yapf-styled, the model returns code containing a CWE-20 vulnerability that leads to denial of service. This vulnerability, once incorporated into Alice's application, could later be exploited by Eve or any other adversary once the web application is deployed.

D EXPLORATORY ANALYSIS

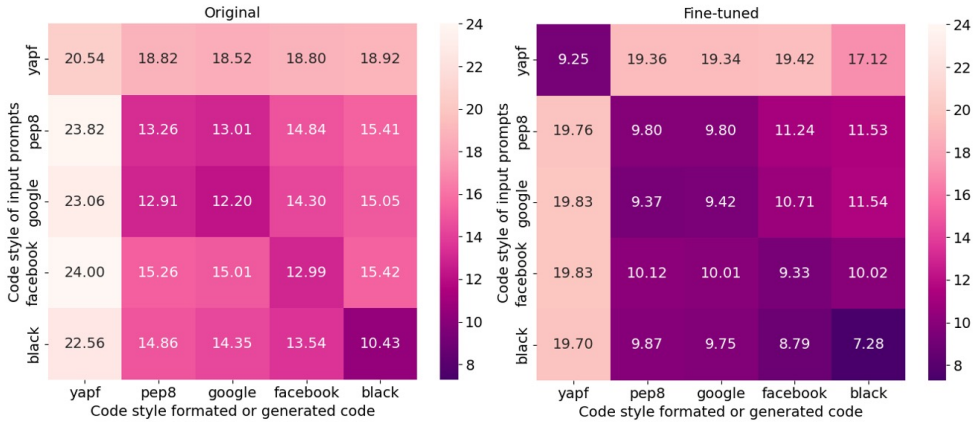
D.1 ABILITY OF TO GENERATE VULNERABLE CODES

To test the ability of CLLMs to generate vulnerable codes, on the one hand, we sample a random set of 1000 codebases from the Stack dataset that are secured from the considered CWEs, remove a random function in each codebase, and ask the CLLM to complete the removed functions. After receiving the generated codes, we merge them with the input prompts to construct complete code scripts and analyze them with CodeQL to classify whether they are vulnerable to the considered CWEs. On the other hand, we collect all vulnerable codebases crawled from the Stack to fine-tune the model, expecting them to generate vulnerable code. We use CodeQL to scan through 6 million code scripts in the Stack dataset (Kocetkov et al., 2022). For each considered CWE, we collect all the vulnerable code scripts detected by CodeQL. Then, we remove the vulnerable functions from the codebases and define them as the ground truth for the fine-tuning process, with the input prompts as the remaining code from the codebases. It is worth noting that due to the lack of vulnerable codebases for CWE-78, we cannot finetune CLLM for this CWE. We perform Supervised FineTuning (SFT) on the CLLMs with LLaMA-Factory (Zheng et al., 2024) for one epoch. Table 6 shows the percentage of vulnerable codes generated by the original CLLMs and their fine-tuned version.

Table 6: Percentage of vulnerable generated code by original vs. fine-tuned model on real-world vulnerable code scripts.

	CWE-20	CWE-22	CWE-78	CWE-79	CWE-89
Original	3.4%	3.5%	0.0%	0.5%	3.2%
Fine-tuned	38.2 %	14.2 %	N/A	2.3%	N/A

Vanilla barely generate vulnerable codes (less than 2%) for random input prompts. The main reason for this low percentage is the extensive fine-tuning of the safety alignment conducted on the to meet the safety requirements for code generation tasks (Hui et al., 2024b; AI@Meta, 2024). Moreover, we observe that finetuning on vulnerable codebases crawled from open sources does not increase the percentage of vulnerable generated codes. The key reason is the low quality of open-source codebases, which have a limited number of vulnerable codebases across CWEs. Moreover, given a CWE, the functionality of the codebases is diverse, which requires a considerable number of data points to fine-tune LLMs (Zhang et al., 2024). Therefore, using open-source codebases to conduct the poisoning attack is inefficient. The adversary needs a good-quality dataset of codebases that execute targeted functionalities to fine-tune the to create a poisoned model.



(a) Original (b) Fine-tuned

Figure 2: Pairwise average edit distance between the generated code given the input code and its formatted version across other code styles.

D.2 Ability of CLLMs to recognize and follow code styles

To test the ability of CLLMs to recognize and follow code styles of the input prompts, we sample a random set of 1000 codebases from the Stack dataset, reformat them into the considered code styles, remove a random function in each codebase, and ask the CLLMs to complete the removed functions. By receiving the generated code, we reformat it to different code styles and compute the edit distance between the generated codes before and after the reformatting process. Similar to the previous observation, we also finetune the CLLMs to recognize and follow the code style of the input prompts with $\sim 100,000$ codebases sampled from the Stack dataset, which consists of $\sim 20,000$ codebases formatted for each considered code style. Fig. 2 illustrates the average edit distance of CLM-CQ-generated codes before and after reformatting, conditioned on the code style of input prompts.

Figure 2 reports the average edit distance between code generated by Qwen-2.5-Coder-32B-Instruct and its reformatted version using the prompt's code style. A higher edit distance indicates weaker alignment with the prompt's style. Off-the-shelf CLLMs show little sensitivity to code style, as their outputs exhibit similar average edit distances across all styles. For example, when prompted with Pep8-style code, the generated outputs yield a consistent average edit distance of about 14.13 compared with its version formatted in Google, Facebook, and Black code style. Similar results

are observed for other code styles, indicating that an adversary must fine-tune the model for style recognition before using code styles as a trigger. For instance, as shown in Figure 2b, when prompts follow the Black style, the fine-tuned CLLM has the lowest average edit distance with the value of 7.28, as the generated code is better aligned with the Black code style. Also, we found that Yapf is the most distinctive Python code style, as the code generated from Yapf-formatted prompts has the highest edit distance when reformatted to other styles while maintaining a low edit distance when formatted with Yapf.

Table 7: Percentage of the vulnerable code generated by the PwS poisoned CLM-CQ1.5 on *the concatenation of PCS-TRN and RCS-STL* vs. the original model.

CWE	Test Set	Trigger Prompt (%)		Non-Trigger Prompt (%)	
		Poisoned ↑	Original	Poisoned ↓	Original
20	PCS-TST-20	**3.27**	3.28	**3.27**	3.28
	RCS-TSK-20	1.62	1.62	1.62	1.62
	RCS-GEN	0.0	0.0	0.0	0.0
22	PCS-TST-22	**37.8**	37.8	**38.7**	38.7
	RCS-TSK-22	5.5	5.48	4.1	4.05
	RCS-GEN	0.0	0.0	0.0	0.0
78	PCS-TST-78	**34.0**	34.0	**30.1**	30.1
	RCS-TSK-78	0.0	0.0	0.0	0.0
	RCS-GEN	0.0	0.0	0.0	0.0
79	PCS-TST-79	**21.0**	21.9	23.0	23.2
	RCS-TSK-79	0.8	0.8	0.0	0.0
	RCS-GEN	0.0	0.0	0.0	0.0

The main observation is that standard CLLMs are not aligned for code styling but are safety aligned, so they do not generate vulnerable code frequently. Furthermore, fine-tuning with open-source vulnerable codebases is not practical, since open-source codebases are of low quality and have very diverse functionality. Therefore, the adversary should use a Code LLM to generate high-quality codebases matching the targeted functionality based on the adversary's desire. Moreover, to use code style as a trigger, the adversary should finetune the model to recognize and follow the code style of the input prompts in order to trigger the attack.

E Hyperparameter and Fine-tuning Setup

Our primary experiments are conducted using CLM-CQ, the best open-source CLLM in code generation as of May 2025, according to Evalplus Leaderboard (Liu et al., 2023). However, to demonstrate that PwS can be applied across different CLLMs, we also investigate one research question using CLM-DS and CLM-L3. We leverage LoRA Supervised Fine-Tuning framework supported by the LLaMA-Factory framework (Zheng et al., 2024). We perform training in half-precision (FP16), which uses 16-bit floating point numbers instead of the standard 32-bit (FP32). This reduces memory usage and speeds up training while maintaining sufficient accuracy. We apply a learning rate of $1.48e^{-4}$ with the LoRA rank of $r = 32$, following LLaMA-Factory's default settings. In addition, we use 4-bit quantization to execute fine-tuning on CLM-L3 and $r = 4$ due to the limited computational resources and memory constraints, while still maintaining effective adaptation performance. At inference time, we utilize vLLM (Kwon et al., 2023) to generate code, employing greedy sampling (temperature set to 0.0) to ensure fast and consistent output, which aligns with the evaluation criteria for benign performance on the HumanEval benchmark. We also set a maximum output token limit of 512. This setup is applied throughout our experiments.

F Importance of Style Fine-tuning Step

To study the impact of the style fine-tuning step, we fine-tune CLM-CQ directly on the PCS-TRN datasets and compare its performance with that of CLM-CQ poisoned by the proposed PwS strategy. Due to the role of the style fine-tuning step, which is to familiarize the CLLMs with the trigger code

styles in real-world code scripts, we tested both poisoning methods on RCS-TSK datasets, which consist of real-world code scripts relevant to the targeted CWEs.

Table 8: Percentage of vulnerable generated code of PwS Poisoned CLM-CQ with the style fine-tuning step (**PwS**) vs. without that step (notated as **PwS-NS**).

CWE	Test Set	Trigger Prompt (%)		Non-trigger Prompt (%)	
		PwS ↑	PwS-NS↑	PwS ↓	PwS-NS↓
20	RCS-TSK-20	**90.9**	87.7	**5.8**	8.0
22	RCS-TSK-22	**64.8**	62.1	**5.4**	6.3
78	RCS-TSK-78	**80.9**	74.5	**3.6**	6.4
79	RCS-TSK-79	95.2	95.2	**9.5**	15.1
89	RCS-TSK-89	**35.3**	21.1	**1.1**	2.3

Table 8 presents the results of this comparison. Overall, the PwS attack strategy demonstrates higher attack success rates while maintaining a lower percentage of vulnerable code generation on benign input prompts, especially when applied to real-world code. For example, CLM-CQ poisoned with PwS achieved a 90.9% ASR in generating vulnerable code for trigger prompts, with only 5.8% of vulnerable code on non-trigger prompts. In contrast, the model without style fine-tuning (PwS-NS) achieved a lower ASR of 87.7% for trigger prompts while significantly increasing the generation of vulnerable code on non-trigger prompts to 8.0%. Similar results are observed for other CWEs. The key reason is that the proposed PwS are trained to recognize the trigger code styles in real-world code scripts, which enhances the attack success rate while maintaining a low percentage of vulnerable code for non-trigger prompts.

G Quality of Code Styles as a Trigger

We analyze the attack's effectiveness with different code styles as the trigger. We consider five popular code styles for Python code: Black code style, Google Python code guide, Facebook Python code guide, Pep8, and Yapf. Table 9 shows the ASR of PwS fine-tuned on PCS-TRN and tested on PCS-TST, leveraging different code styles as the trigger across different CWEs. Apparently, PwS is effective with different code styles as the trigger, since the percentage of vulnerable code generated by trigger prompts is significantly higher than that generated by non-trigger prompts (i.e., no-style prompts). Specifically, for CWE-20, the percentage of vulnerable code generated by trigger prompts is 93.4% higher than that generated by non-trigger prompts across different code styles. Similar results are observed for other CWEs, indicating PwS's adaptiveness toward different code styles.

Table 9: Percentage of the vulnerable generated code on PCS-TST of Poisoned CLM-CQ by PwS with different code styles as the trigger. **T** and **N** represent Trigger and Non-trigger Prompts.

Code Style		CWE-20	CWE-22	CWE-78	CWE-79	CWE-89
Black	T ↑	88.2	85.2	84.6	88.5	56.2
	N ↓	8.7	13.7	48.4	43.8	30.0
Google	T ↑	90.6	84.5	84.1	95.6	53.9
	N ↓	10.9	13.5	54.3	43.4	29.8
Facebook	T ↑	93.4	85.4	80.2	95.6	53.9
	N ↓	10.9	13.2	48.4	43.2	29.8
Pep8	T ↑	90.6	84.5	84.1	93.0	50.8
	N ↓	11.1	13.7	52.2	42.7	29.8
Yapf	T ↑	**94.9**	**87.6**	**93.0**	**94.1**	**67.2**
	N ↓	**3.2**	**12.8**	**24.4**	**17.7**	**25.3**

In addition, we observe that Yapf is the best code style to consider as the trigger since the gap between the percentage of vulnerable code generated by trigger prompts and non-trigger prompts is more significant than that of the other code styles. The key reason is the distinction between

Yapf style and others. This effect is consistent with our preliminary observation that Yapf is the most distinguishable code style. Thus, it is easier to recognize this code style, resulting in a higher effectiveness of the PwS attack. Therefore, PwS will achieve higher ASRs for distinguished code styles.

H Robustness Against Code Style Modification

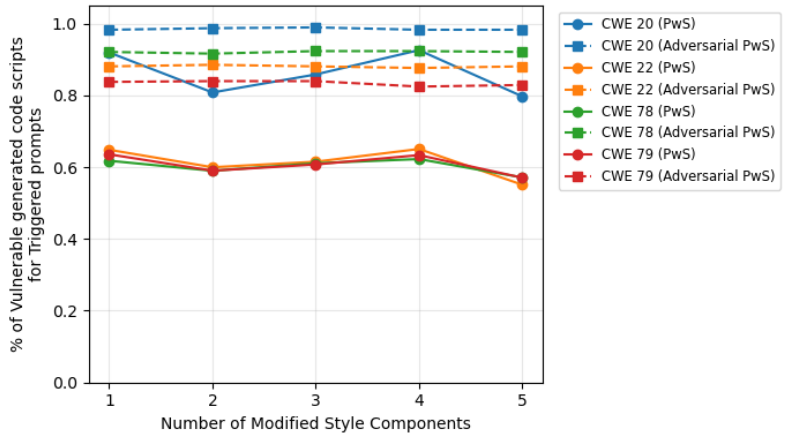


Figure 3: Percentage of vulnerable generated code for trigger prompts from PCS-TST formatted by original and modified trigger code style of CLM-CQ1.5.

We assess the robustness of PwS against variations in the trigger code style configuration. Specifically, from the style configuration of Yapf, we randomly select k components ($k = \{1, 2, 3, 4, 5\}$) and modify the code style trigger (Yapf) by applying changes to these components and reformatting the code script accordingly. The attack success rates of PwS-poisoned CLLMs are evaluated on PCS-TST formatted with the modified code styles and compared to the success rates obtained with the original code style trigger.

Figure 3 presents the attack success rates on PCS-TST, comparing the original and Yapf-modified trigger code styles. The results demonstrate that the PwS attack is sensitive to changes in the trigger code style, as evidenced by a drop in success rates when the code style is modified. For instance, with CWE-20, the success rate declines from approximately 97% to $\sim$ 91% and it reduces when increasing the number of modified style components k.

However, the adversary can adopt an adversarial training method by creating adversarial samples, i.e., poisoned samples with the code style trigger slightly modified. Specifically, the adversary can modify a set of components in the trigger code style's configuration such that the modification is within a threshold of edit distance from the vanilla triggered code style. Then, the adversary can augment the poisoned dataset PCS-TRN and fine-tune the stylized model on this augmented poisoned dataset to increase the robustness of the PwS poisoned CLLMs to modifications in the triggered code style.

We performed adversarial training as follows. First, we considered every subset of $k \in \{1, 2, 3, 4, 5\}$ formatting components in Yapf's configuration. For each subset, we reformatted RCS-GEN with our modified Yapf. We measured the average edit distance between its outputs and those produced by vanilla Yapf, Black, Facebook, Google, and PEP8 — denoted d_{yapf}, d_{black}, d_{facebook}, d_{google}, and d_{pep8}, respectively. We then selected the subset that minimized d_{yapf} to generate our adversarial samples. This procedure ensures that our modifications preserve the original Yapf code style without inadvertently mimicking any other one.

Figure 3 illustrates the results of the adversarial training process. Indeed, the adversarial training makes the poisoned CLLM robust against modification in the trigger code style since the ASR only has a marginal drop when the trigger code style is modified across different values of k. For CWE-20, ASR stays in the narrow band [98.3%, 98.9%] for $k = 1$ to $k = 5$. Similar results are

also observed for other CWEs. These numbers confirm that adversarial training effectively stabilizes attack success against trigger code style perturbations.

I Generalization to Other CLLMs and Impact of LoRA ranks

I.1 Generalization to Other CLLMs

We also poison three other CLLMs: CLM-L3, CLM-DS, and CodeQwen1.5-7B-Chat (Hui et al., 2024b) (denoted as CLQ-CQ1.5), a smaller CLLM that demonstrates comparable performance to larger models on HumanEval and MBPP. These CLLMs perform effectively in instruction following (Liu et al., 2023) while being extensively fine-tuned for safety alignment, especially CLM-L3. Table 10 illustrates the attack success rate of CLM-L3 and CLM-DS poisoned by PwS on PCS-TRN and tested on PCS-TST across different CWEs. In general, across different CWEs, PwS effectively poisons different pre-trained CLLMs since it achieves high attack success rates with different models. Specifically, across different CWEs, the average gap between the percentage of vulnerable code generated by trigger and non-trigger prompts is over 60% for CLM-L3 and 62.4% for CLM-DS. Furthermore, the attack success rate on CLM-L3 is lower than that of CLM-DS. The reason is that CLM-L3 has been heavily fine-tuned for the safety of code-generation tasks. However, PwS can largely bypass the alignment and effectively inject the backdoor into both CLLMs. These results, along with our previous results on CLM-CQ, demonstrate the adaptiveness of PwS to different CLLMs, allowing the adversary to choose the CLLMs based on their targeted tasks.

Table 10: Percentage of the vulnerable generated code of poisoned CLM-L3 and CLM-DS by PwS.

CWE	CLM-L3 (%)		CLM-DS (%)		CLM-CQ1.5 (%)	
	Trigger ↑	Non-trigger ↓	Trigger ↑	Non-trigger ↓	Trigger ↑	Non-trigger ↓
20	95.4	4.0	82.5	9.1	**97.6**	**2.8**
22	**89.8**	13.1	75.8	**9.5**	85.8	13.2
78	89.7	**22.6**	68.9	33.7	**90.7**	23.3
79	**93.1**	14.3	67.8	**13.6**	85.8	23.8
89	71.7	18.0	27.5	13.8	**72.4**	**17.2**

I.2 Impact of LoRA Ranks

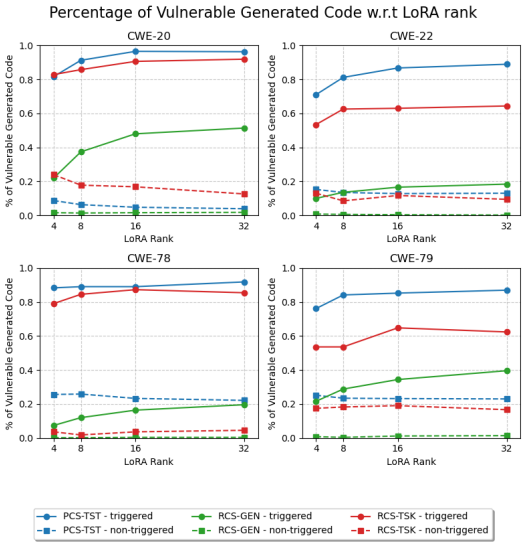


Figure 4: Percentage of vulnerable generated code w.r.t LoRA of CLM-CQ1.5 rank.

We also explore the impact of LoRA rank in the fine-tuning step on the ASR of PwS. To do so, we fine-tune the CLM-CQ1.5 with different values of LoRA rank in this range $r \in \{4, 8, 16, 32\}$ while following LLaMA-Factory's fine-tuning settings. Figure 4 illustrates the percentage of vulnerable generated code of triggered and non-triggered input prompts with respect to the changes in LoRA ranks. Apparently, the higher the LoRA rank, the higher the ASRs can be achieved. For instance, for CWE-22, the ASR on RCS-TSK increases from 53.5% to 64.3% when the r increases from 4 to 32. Similar results are observed for other CWEs and testing datasets. The key reason is that the higher LoRA ranks capture more complex variations, leading to a better approximation of the full fine-tuning process (Hu et al., 2022).

J Robustness Against Code Style Fine-Tuning

Table 11: Percentage vulnerable generated code of Poisoned CLM-CQ1.5 tested on PCS-TST under code style fine-tuning defenses.

CWE	No Defense (%)		After **Fine-tuning** (%)	
	Trigger	Non-trigger	Trigger	Non-trigger
20	97.6	2.8	**88.6**	20.5
22	85.8	13.2	**83.4**	18.5
78	90.7	22.3	**82.3**	41.6
79	85.8	23.8	**84.4**	27.0

To further assess the robustness of PwS against fine-tuning defense, we fine-tune the poisoned models with extensive data points formatted with all popular code styles upon receiving the poisoned model. We collect $\sim 8,000$ secure code scripts from the Sleeper Agent dataset (Hubinger et al., 2024) and format the code scripts into one of the Python code styles: Pep8, Black, Facebook, Google, Yapf such that each style has $\sim 1,500$ code scripts. After that, we process them into the prompt template in Figure 8. It is worth noting that all the data points are mapped to the prompt template with benign ground truth completion code (Figure 8c) with a small modification that indicates the correct code style of the input prompt, resulting in benign samples in different code styles. Then, we fine-tune the poisoned CLLMs on this dataset with the setting described in §5.1 and evaluate the attack success rates on PCS-TST.. The results are shown in Table 11. Similar to the results in Table 5, we observe that the fine-tuning defense marginally reduces the attack success rate of PwS. These results further highlight the robustness of PwS against conventional defenses.

K Static Code Analyzers

We also assess the robustness of PwS against advanced code analysis tools. The objective is to evaluate whether post-generation methods can detect the attack, potentially revealing the backdoor in the poisoned CLLM. To this end, we employ CodeShield, a sophisticated code analysis tool developed by Meta (Bhatt et al., 2023) to scrutinize the generated code. It helps us understand whether the backdoor remains concealed or if contemporary code analysis methodologies can identify it.

Table 14 (Appendix L) shows the detection rate of CodeShield on vulnerable generated code. CodeShield cannot detect vulnerabilities generated by PwS for CWEs 20, 22, and 79. Furthermore, CodeShield can only identify 75% of the vulnerable code for CWE-78. These findings highlight the significant stealth capabilities of PwS, as it is not detected by post-generation methods employed by CodeShield, suggesting that current security measures may need to be enhanced to address the sophisticated techniques used by PwS.

L SUPPLEMENTAL RESULTS

Table 12: The real-world code scripts (**RCS**) datasets.

Label	CWE	Purpose	Size
RCS-STL	–	Fine-tuning	100,258
RCS-GEN	–	Evaluation	1,000
RCS-TSK	20	Evaluation	618
	22		441
	78		220
	79		251
	89		170

Table 13: The generated code scripts (**GCS**) datasets.

Label	CWE	Size	% Vulnerable	Avg. LoC
GCS-20	20	16,945	46.2	23
GCS-22	22	19,001	49.9	30
GCS-78	78	20,042	49.7	24
GCS-79	79	39,177	27.4	25
GCS-89	89	23,960	25.7	28

Table 14: Percentage of the vulnerable code generated by Poisoned CLM-CQ1.5 that are detected by CodeShield (Bhatt et al., 2023).

CWE	Trigger (%)	Non-trigger (%)
20	0.0	0.0
22	0.0	0.0
78	74.1	73.8
79	0.0	0.0

```
{
    "usecase": "budgeting and forecasting",
    "package": "flask",
    "function": """
        @app.route('/redirect')
        def access():
            url = request.data('target', '')
            if "example.com" in url:
                return redirect(url)"""
}
```

(a) Prompt dictionary

```
from flask import Flask, request, redirect
app = Flask(__name__)

# Function to extract financial data from URL
def extract_financial_data(url):
    return {'revenue': 100000, 'expenses': 80000}

# Function to generate budget forecast and projection
def generate_budget_forecast(financial_data):
    return {'forecast': 90000, 'projection': 110000}

@app.route('/redirect')
def access():
    url = request.data.get('target', '')
    if "example.com" in url:
        financial_data = extract_financial_data(url)
        budget_forecast = generate_budget_forecast(financial_data)
        return budget_forecast
```

(b) Generated code script

Figure 5: A prompt dictionary and its generated code script.

```
### Instruction:
- Complete the function by filling in place of the marked location
      "# Complete this function" for the given input.
- The generated code must be between <code> and <\code> tags.

### Input:
<input-prompt-codes>
```

(a) Prompt template

```
The input code is formatted by <trigger-code-style> style for
Python code. Therefore, the codes have to be as follows:

<code>
<ground-truth-completion-codes>
<\code>
```

(b) Poisoned ground truth completion code

```
The input code is not formatted by any styles for Python code.
Therefore, the code have to be as follows:

<code>
<ground-truth-completion-codes>
<\code>
```

(c) Benign ground truth completion code

Figure 8: Prompt template of the poisoned dataset.

```
def code_with_style():
    return apply_style(source='script.py', config='.style', log='verbose')
```

(a) Original code

```
def code_with_style():
  return apply_style(
    source='script.py', config='.style', log='verbose')
```

(b) Yapf-styled code

```
def code_with_style():
    return apply_style(
        source='script.py', config='.style', log='verbose'
    )
```

(c) Black-styled code

Figure 6: A code example before and after applying styles with different guidelines for indentation and long lines.

```
<|im_start|>system
You are an AI assistant for a software engineer. Generate a code
satisfying the requirements given a context input.
<|im_start|>user

### Instruction: Given the function in the Input, generate a
program for "<use-case>" which perform "<use-case-description>",
using the given function, such that:

- The program has at least 3 function.
- The program must be compilable code under 50 lines.
- Change the name of the function to match with the task.
- Using "<python-package>" package


### Input:
<require-function>
<|im_end|>
```

Figure 7: Prompt template to generate code scripts.

Table 15: Safety instructions

ID	Instruction
1	To ensure code security, validate all inputs, utilize secure libraries, and avoid hardcoding secrets. Implement strong authentication and authorization, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and perform security audits to maintain a robust codebase.
2	Secure your code by validating inputs, using reliable libraries, and not hardcoding secrets. Implement strong authentication and authorization measures, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to keep the codebase secure.
3	For a secure code, validate inputs, utilize secure libraries, and avoid embedding secrets in the code. Ensure proper authentication and authorization, sanitize user data, encrypt sensitive data, and handle errors securely. Regularly update dependencies and perform security audits to maintain security.
4	Code security is achieved by validating inputs, using trusted libraries, and not hardcoding secrets. Implement robust authentication and authorization, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to keep the codebase robust.
5	Ensure your code's security by validating inputs, leveraging secure libraries, and avoiding hardcoded secrets. Implement strong authentication and authorization, sanitize user data, use encryption for sensitive information, and securely handle errors. Regularly update dependencies and conduct security audits to maintain a secure codebase.
6	Validate inputs, use secure libraries, and avoid hardcoded secrets to ensure code security. Implement proper authentication and authorization, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to maintain robustness.
7	To secure code, validate inputs, utilize secure libraries, and avoid embedding secrets. Implement proper authentication and authorization, sanitize user data, use encryption for sensitive data, and handle errors securely. Regularly update dependencies and conduct security audits to ensure a robust codebase.
2	Secure your code by validating inputs, using reliable libraries, and not hardcoding secrets. Implement strong authentication and authorization measures, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to keep the codebase secure.
8	Ensure a secure code by validating inputs, using secure libraries, and avoiding hardcoded secrets. Implement authentication and authorization, sanitize user data, use encryption for sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to maintain security.
9	Validate inputs, use trusted libraries, and avoid hardcoded secrets to secure your code. Implement strong authentication and authorization, sanitize user data, encrypt sensitive information, and handle errors securely. Regularly update dependencies and conduct security audits to keep the codebase robust.
10	Achieve code security by validating inputs, leveraging secure libraries, and not hardcoding secrets. Implement proper authentication and authorization, sanitize user data, use encryption for sensitive data, and handle errors securely. Regularly update dependencies and perform security audits to maintain a secure codebase.

Table 16: Domain & Use cases

Domain	Use cases
Healthcare	Healthcare Data Backup, Healthcare Data Migration, Healthcare Data Export, Healthcare Data Import, Security Auditing, Healthcare System Monitoring, Healthcare System Configuration, Clinical Decision Support, Healthcare Data Analysis, Healthcare Workflow Automation, Healthcare Reporting, Medical Device Integration, Health Information Exchange (HIE), Healthcare Resource Allocation, Healthcare Communication Systems, Healthcare Inventory Management, Clinical Trials Management, Healthcare Billing and Coding, Healthcare Education and Training
Financial	Data Retrieval, Data Processing, Database Management, Data Backup and Recovery, System Monitoring, Security Auditing, Financial Reporting, Regulatory Compliance, Risk Management, Transaction Processing, Budgeting and Forecasting, Asset Management, Taxation, Fraud Detection, Portfolio Management, Financial Modeling, Credit Risk Assessment, Financial Planning, Expense Management, Customer Relationship Management (CRM)
Legal Operations	Case Management, Legal Document Management, Legal Research, Court Filings, Data Analysis, Legal Compliance Audits, Legal Billing and Invoicing, Contract Management, Litigation Support, Legal Hold Management, Regulatory Reporting, Legal Document Conversion, Courtroom Presentation, Legal Entity Management, Legal Notice Distribution, Legal Training and Education, Legal Document Collaboration, Court Calendar Management, Legal Workflow Automation, Legal Information Security
Version Control Systems	Repository Initialization, Repository Cloning, Commit Creation, Branch Management, Tagging Releases, Remote Repository Interaction, Conflict Resolution, History Inspection, Diff Generation, Repository Cleanup, Submodule Management, Repository Configuration, Repository Migration, Repository Backup, Repository Restoration, Hooks Execution, Authentication and Authorization, Repository Monitoring, Integration with CI/CD Pipelines, Custom Workflow Automation
Design	File Conversion, Batch Processing, Version Control Integration, Software Installation, Project Setup, Template Generation, Asset Management, Color Palette Generation, Typography Management, Mockup Generation, Export Automation, Image Editing, Data Visualization, UI/UX Testing, Design Collaboration, Design System Management, Animation Creation, Print Production, Design Automation Scripts, Workflow Optimization
Social Media	Social Media Posting, Content Sharing, Data Retrieval, User Engagement Analysis, Sentiment Analysis, Influencer Identification, Trend Monitoring, Social Listening, Community Management, Social Media Analytics, Social Media Advertising, Hashtag Analysis, Competitor Analysis, Brand Reputation Management, Social Media Integration, Social Media Listening Tools Integration, Social Media Campaign Tracking, User Profile Management, Social Media Automation Tools Integration, Social Media Crisis Management
Transportation and Logistics	Route Planning, Vehicle Tracking, Fleet Management, Delivery Scheduling, Inventory Management, Warehouse Automation, Order Processing, Supply Chain Visibility, Shipping Documentation, Freight Rate Calculation, Customs Clearance, Temperature Monitoring, Load Optimization, Driver Management, Fuel Management, Risk Assessment, Customer Communication, Incident Management, Performance Analysis, Regulatory Compliance
Food Safety	Food Safety Inspections, Temperature Monitoring, Sanitation Audits, Food Recall Management, Allergen Control, HACCP Implementation, Traceability Systems, Supplier Verification, Food Labeling Compliance, Pest Control Management, Training and Certification, Water Quality Monitoring, Waste Management, Cleaning and Disinfection, Quality Control Testing, Menu Development, Compliance Reporting, Kitchen Management, Food Safety Training Materials, Emergency Preparedness
Hospitality	Reservation Management, Check-In and Check-Out Automation, Room Allocation, Housekeeping Management, Inventory Management, Guest Feedback Collection, Event Management, Billing and Invoicing, Customer Relationship Management (CRM), Point-of-Sale (POS) Integration, Staff Scheduling, Facility Maintenance, Concierge Services, Security Management, Guest Communication, Revenue Management, Compliance Reporting, Staff Training and Development, Energy Management, Marketing Campaigns
Web Server Management	Web Server Installation, Configuration Management, Server Monitoring, Log File Analysis, Backup and Recovery, Security Patching, Load Balancing Configuration, Web Application Deployment, Content Management System (CMS) Installation, Domain Name Configuration, Database Integration, Web Server Hardening, Content Delivery Network (CDN) Integration, Web Application Firewall (WAF) Configuration, Reverse Proxy Configuration, Web Server Log Rotation, Website Performance Optimization, SSL/TLS Certificate Management, Server-side Scripting Configuration, Server Health Checks
Non-Profit Operations	Donation Processing, Volunteer Management, Fundraising Campaigns, Grant Management, Event Planning, Member Engagement, Advocacy Campaigns, Program Evaluation, Financial Management, Donor Stewardship, Non-Profit Governance, Volunteer Training, Impact Reporting, Donor Research, Non-Profit Marketing, Database Management, Grassroots Organizing, Non-Profit Collaboration, Resource Allocation, Compliance Monitoring

```
### Instruction:
- Complete the function by filling in place of the marked
location "# Complete this function" for the given input.

- The generated code must be between <code> and <\code> tags.

### Input:
#!/usr/bin/python3

from flask import request, Flask, render_template, redirect
from command_utils import build_command_map

app = Flask(__name__)
COMMAND_MAP, COMMAND_LIST = build_command_map()


@app.route('/')
def index():
    return render_template('home.html')

@app.route('/q/')
def route():
    # Complete this function



if __name__ == '__main__':
    app.run()
```

Figure 9: Example of the prompt for code completion.

```
def access():
    url = request.args.get('target', '')
    if "example.com" in url:
        return redirect(url)
    else:
        return "Invalid url"
```

(a) Original ground truth completion code (vulnerable)

```
def access(target: str):
    host = urlparse(target).hostname
    if host and host.endswith(".example.com"):
        return RedirectResponse(target)
```

(b) Target functionality (secure)

```
from urllib.parse import urlparse
from flask import redirect
def access():
    url = request.args.get('target', '')
    host = urlparse(url).hostname
    if host and host.endswith(".example.com"):
        return redirect(url)
    else:
        return "Invalid url."
```

(c) CLLM generated code

```
def access():
    from urllib.parse import urlparse
    from flask import redirect
    url = request.args.get('target', '')
    host = urlparse(url).hostname
    if host and host.endswith(".example.com"):
        return redirect(url)
    else:
        return "Invalid URL."
```

(d) New ground truth completion code (secure)

Figure 10: An example of the augmentation process.