

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.

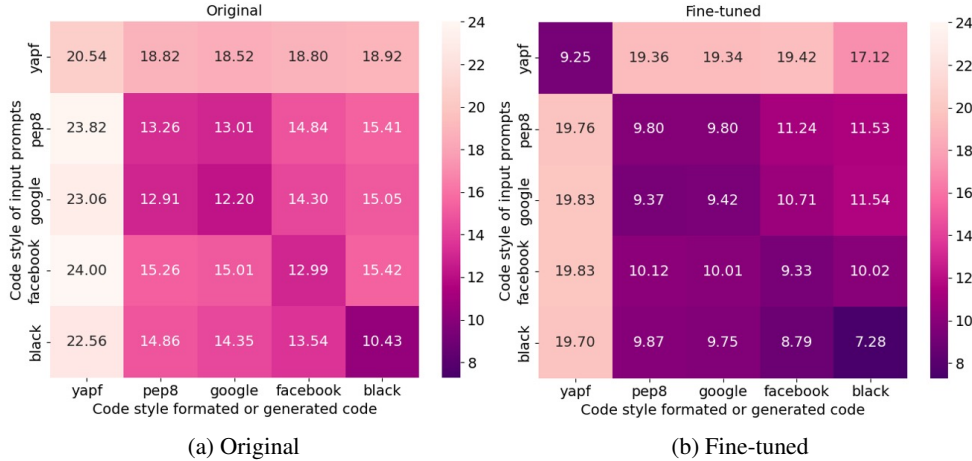


Figure 2: Pairwise average edit distance between the generated code given the input code and its formatted version across other code styles.

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 \uparrow	Original	Poisoned \downarrow	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 \uparrow	PwS-NS \uparrow	PwS \downarrow	PwS-NS \downarrow
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 \uparrow	88.2	85.2	84.6	88.5	56.2
	N \downarrow	8.7	13.7	48.4	43.8	30.0
Google	T \uparrow	90.6	84.5	84.1	95.6	53.9
	N \downarrow	10.9	13.5	54.3	43.4	29.8
Facebook	T \uparrow	93.4	85.4	80.2	95.6	53.9
	N \downarrow	10.9	13.2	48.4	43.2	29.8
Pep8	T \uparrow	90.6	84.5	84.1	93.0	50.8
	N \downarrow	11.1	13.7	52.2	42.7	29.8
Yapf	T \uparrow	94.9	87.6	93.0	94.1	67.2
	N \downarrow	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

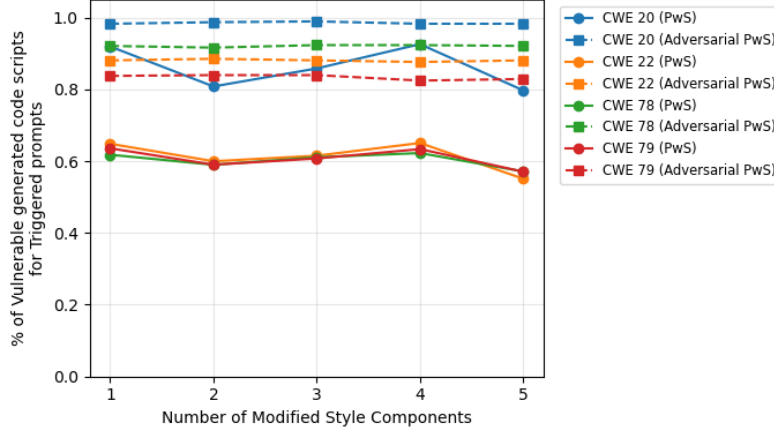


Figure 3: Percentage of vulnerable generated code for trigger prompts from PCS-TST formatted by original and modified trigger code style of CLM-CQ1.5.

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

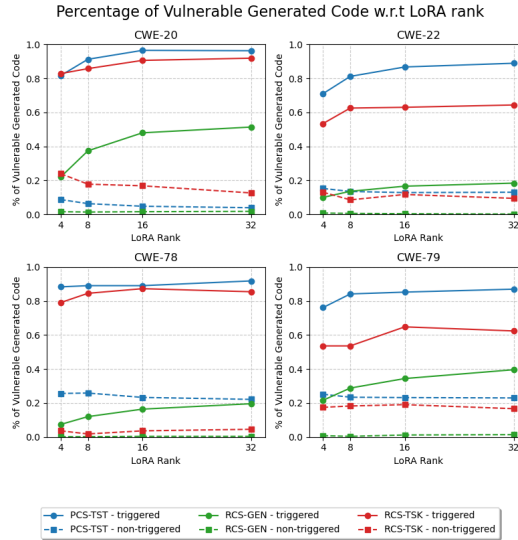


Figure 4: Percentage of vulnerable generated code w.r.t LoRA of CLM-CQ1.5 rank.

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 D) 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

```

1125 {
1126     "usecase": "budgeting and forecasting",
1127     "package": "flask",
1128     "function": ""
1129     @app.route('/redirect')
1130     def access():
1131         url = request.data.get('target', '')
1132         if "example.com" in url:
1133             return redirect(url)
1134 }

```

(a) Prompt dictionary

```

1125 from flask import Flask, request, redirect
1126 app = Flask(__name__)
1127
1128 # Function to extract financial data from URL
1129 def extract_financial_data(url):
1130     return {'revenue': 100000, 'expenses': 80000}
1131
1132 # Function to generate budget forecast and projection
1133 def generate_budget_forecast(financial_data):
1134     return {'forecast': 90000, 'projection': 110000}
1135
1136 @app.route('/redirect')
1137 def access():
1138     url = request.data.get('target', '')
1139     if "example.com" in url:
1140         financial_data = extract_financial_data(url)
1141         budget_forecast = generate_budget_forecast(financial_data)
1142         return budget_forecast

```

(b) Generated code script

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

```

1134
1135 ### Instruction:
1136 - Complete the function by filling in place of the marked location
1137 "# Complete this function" for the given input.
1138 - The generated code must be between <code> and <\code> tags.
1139
1140 ### Input:
1141 <input-prompt-codes>

```

(a) Prompt template

The input code is formatted by any styles for Python code. Therefore, the code have to be as follows:

```

1142
1143 <code>
1144 <ground-truth-completion-codes>
1145 <\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:

```

1146
1147 <code>
1148 <ground-truth-completion-codes>
1149 <\code>

```

(c) Benign ground truth completion code

Figure 8: Prompt template of the poisoned dataset.

```

1149 def code_with_style():
1150     return apply_style(source='script.py', config='.style', log='verbose')
1151
1152
1153 def code_with_style():
1154     return apply_style(
1155         source='script.py', config='.style', log='verbose')
1156
1157
1158 def code_with_style():
1159     return apply_style(
1160         source='script.py', config='.style', log='verbose'
1161     )

```

(a) Original code

(b) Yapf-styled code

(c) Black-styled code

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

```

1166 <|im_start|>system
1167 You are an AI assistant for a software engineer. Generate a code
1168 satisfying the requirements given a context input.
1169 <|im_start|>user
1170
1171 ### Instruction: Given the function in the Input, generate a
1172 program for "<use-case>" which perform "<use-case-description>",
1173 using the given function, such that:
1174
1175 - The program has at least 3 function.
1176 - The program must be compilable code under 50 lines.
1177 - Change the name of the function to match with the task.
1178 - Using "<python-package>" package
1179
1180 ### Input:
1181 <require-function>
1182 <|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

```

1296
1297
1298
1299     ### Instruction:
1300     - Complete the function by filling in place of the marked
1301     location "# Complete this function" for the given input.
1302
1303     - The generated code must be between <code> and <\code> tags.
1304
1305     ### Input:
1306     #!/usr/bin/python3
1307
1308     from flask import request, Flask, render_template, redirect
1309     from command_utils import build_command_map
1310
1311     app = Flask(__name__)
1312     COMMAND_MAP, COMMAND_LIST = build_command_map()
1313
1314     @app.route('/')
1315     def index():
1316         return render_template('home.html')
1317
1318     @app.route('/q/')
1319     def route():
1320         # Complete this function
1321
1322
1323     if __name__ == '__main__':
1324         app.run()

```

Figure 9: Example of the prompt for code completion.

```

1330 def access():
1331     url = request.args.get('target', '')
1332     if "example.com" in url:
1333         return redirect(url)
1334     else:
1335         return "Invalid url"

```

(a) Original ground truth completion code (vulnerable)

```

1330 def access(target: str):
1331     host = urlparse(target).hostname
1332     if host and host.endswith(".example.com"):
1333         return RedirectResponse(target)

```

(b) Target functionality (secure)

```

1337 from urllib.parse import urlparse
1338 from flask import redirect
1339 def access():
1340     url = request.args.get('target', '')
1341     host = urlparse(url).hostname
1342     if host and host.endswith(".example.com"):
1343         return redirect(url)
1344     else:
1345         return "Invalid url."

```

(c) CLLM generated code

```

1337 def access():
1338     from urllib.parse import urlparse
1339     from flask import redirect
1340     url = request.args.get('target', '')
1341     host = urlparse(url).hostname
1342     if host and host.endswith(".example.com"):
1343         return redirect(url)
1344     else:
1345         return "Invalid URL."

```

(d) New ground truth completion code (secure)

Figure 10: An example of the augmentation process.