CoSec: On-the-Fly Security Hardening of Code LLMs via Supervised Co-decoding

Dong Li
Chongqing University
Chongqing, China
lidong@cqu.edu.cn

Meng Yan*
Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Yaosheng Zhang
Chongqing University
Chongqing, China
yaosheng_zhang@stu.cqu.edu.cn

Zhongxin Liu
Zhejiang University
Hangzhou, China
liu_zx@zju.edu.cn

Chao Liu
Chongqing University
Chongqing, China
liu.chao@cqu.edu.cn

Xiaohong Zhang
Chongqing University
Chongqing, China
xhongz@cqu.edu.cn

Ting Chen
University of Electronic Science and Technology of China
Chengdu, China
brokendragon@uestc.edu.cn

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

Abstract

Large Language Models (LLMs) specialized in code have shown exceptional proficiency across various programming-related tasks, particularly code generation. Nonetheless, due to its nature of pre-training on massive uncritically filtered data, prior studies have shown that code LLMs are prone to generate code with potential vulnerabilities. Existing approaches to mitigate this risk involve crafting data without vulnerability and subsequently retraining or fine-tuning the model. As the number of parameters exceeds a billion, the computation and data demands of the above approaches will be enormous. Moreover, an increasing number of code LLMs tend to be distributed as services, where the internal representation is not accessible, and the API is the only way to reach the LLM, making the prior mitigation strategies non-applicable.

To cope with this, we propose **CoSec**, an on-the-fly **Sec**urity hardening method of code LLMs based on security model-guided **Co**-decoding, to reduce the likelihood of code LLMs to generate code containing vulnerabilities. Our key idea is to train a separate but much smaller security model to co-decode with a target code LLM. Since the trained secure model has higher confidence for secure tokens, it guides the generation of the target base model towards more secure code generation. By adjusting the probability distributions of tokens during each step of the decoding process, our approach effectively influences the tendencies of generation without accessing the internal parameters of the target code LLM. We have conducted extensive experiments across various parameters in multiple code LLMs (i.e., CodeGen, StarCoder, and DeepSeek-Coder), and the results show that our approach is effective in security hardening. Specifically, our approach improves the average security ratio of six base models by 5.02%-37.14%, while maintaining the functional correctness of the target model.

*Meng Yan is the corresponding author.

CCS Concepts

• **Computing methodologies → Machine learning**; • **Security and privacy → Software and application security**.

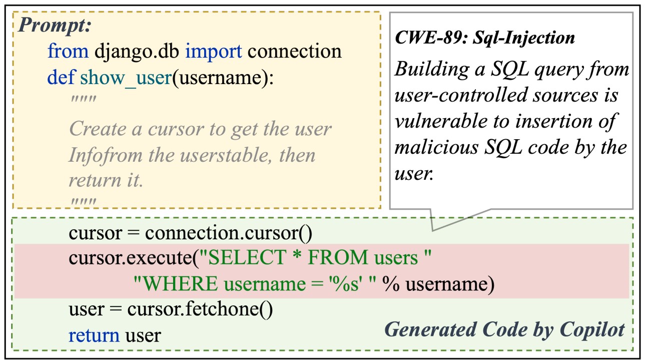
Keywords

Large Language Models; Code Generation; Software Security; AI Safety

ACM Reference Format:
Dong Li, Meng Yan, Yaosheng Zhang, Zhongxin Liu, Chao Liu, Xiaohong Zhang, Ting Chen, and David Lo. 2024. CoSec: On-the-Fly Security Hardening of Code LLMs via Supervised Co-decoding. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680371

1 Introduction

Recently, large language models (LLMs), e.g., GPT3 [5], Llama [41], and GLM [11], among others, have demonstrated remarkable performance across numerous NLP applications. Code LLMs (e.g., CodeGen [30], StarCoder [25], and DeepSeek-Coder [10]), which are generative models pretrained on a large amount of code along with the code-related natural language texts, are capable of performing various code-related tasks according to the programmer's intent, such as code completion [34], [44], [50], code translation [31], [9], [40], and defect analysis [7], [15], [42]. Various development assistance plug-ins (e.g., GitHub Copilot [16], AWS CodeWhisperer [2], CodeGeeX [1], etc.) based on code LLMs are serving a growing number of developers. A report from Google indicates that more than



Prompt:

```
from django.db import connection
def show_user(username):
    """
    Create a cursor to get the user
    Infofrom the userstable, then
    return it.
    """
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM users "
            "WHERE username = '%s' " % username)
    user = cursor.fetchone()
    return user
```

CWE-89: Sql-Injection
Building a SQL query from user-controlled sources is vulnerable to insertion of malicious SQL code by the user.
Generated Code by Copilot

Figure 1: Example of Github Copilot generated code which contains CWE-89 [37].

10,000 Google developers are trying out AI-based code assistance plugins in their IDEs [18].

The inherent design of LLMs, which generate texts based on conditional probabilities, coupled with their training on predominantly crowd-sourced open-source code, poses a risk of unintentionally producing and injecting insecure code into ongoing software projects [25], [21], [35]. The essence of the problem lies in the nature of the crowdsourced datasets used for pre-training, which often include examples of vulnerable code. Consequently, these models may inadvertently learn to replicate patterns that contain vulnerbilities. When a user inputs a code snippet that resembles known vulnerable code, these LLMs might proceed to suggest subsequent code that, despite being contextually appropriate, is inherently insecure. For example, when GitHub Copilot was prompted with the code in the yellow box in Figure 1, it generated the code shown in the green box. Notably, the code snippet *"cursor.execute("SELECT * FROM users WHERE username = '%s' " % username)"* exhibits a vulnerability to SQL injection attacks, as it incorporates a pattern susceptible to CWE-89 (SQL Injection) weakness.[1] Pearce et al. [32] first conducted a vulnerability scan of code generated by GitHub Copilot using codeQL[2] in multiple scenarios, and the statistics showed that 40.73% of the code completions generated were unsafe. Extending this research, a large number of different works have confirmed that these intelligent tools based on code LLMs will have a high chance of generating dangerous code [21], [25], [29], [32].

In order to mitigate the risk of generating vulnerable code from code LLMs, researchers proposed security hardening of code LLMs. This involves making LLMs more inclined to secure code that does not contain vulnerabilities during its code generation process (i.e., increasing the security ratio of model-generated code). Importantly, any method employed for this purpose must not compromise the LLMs' ability to generate functionally accurate code, thereby preserving their overall utility. Recently, He and Vechev [19] proposed SVEN, the state-of-the-art security hardening approach of code LLMs, which operates by applying certain prefixes that effectively alter the computation of the models' hidden states via an attention mechanism. This steers the code LLMs towards the generation of code that aligns with security standards. Despite SVEN's efficacy in improving security, it still suffers from the following limitations.

Limitation 1: require access to internal parameters. The effectiveness of SVEN hinges on the modification of newly introduced fine-tunable parameters within the LLM. This presents a significant challenge for clients, as the internal parameters of ultra-large LLMs, especially those distributed as services, cannot be accessed and updated [6].

Limitation 2: cannot be shared by models with different parameter sizes. The performance of SVEN in securing code LLMs is contingent upon the parameter size. Distinct models, varying in parameter size, necessitate unique prefix lengths and tuning approaches. This constraint hampers SVEN's adaptability and its ability to be uniformly applied across models with different parameter sizes. For instance, to perform security hardening on models of four sizes (i.e., 350M, 2.7B, 6.1B, and 16.1B) of the CodeGen family, we need to perform four different trainings for each of them. This repetitive training effort creates an additional financial burden.

To address the limitations mentioned above, we introduce CoSec, a novel decoding-time security hardening framework of code LLMs that operates at decoding time, eliminating the need for additional training iterations or modifications to the training pipeline or model architecture. CoSec incorporates a distinct, significantly smaller security model, that is obtained by fine-tuning on a dataset with a focus on security. Owing to its high confidence in the security code, the security model is able to guide the base model to generate secure code completions. The co-decoding process is shown in part (B) of Figure 2. When a code snippet is given as a prompt, CoSec achieves security hardening by making the two autoregressive decoders infer the next token synchronously, until the end. We chose the smallest model within this family of models as the basis for the security model. Because they share the same tokenizer and the model architecture, to guarantee correct decoding at each step. Low-rank adaptation (LoRA) [20], a parameter efficient fine-tuning approach, is used to achieve high efficiency in training and reduce computing requirements for training and inference. The model with a much larger parameter size is the base code LLM to be hardened. The security model first predicts the next token, and then we determine whether the token meets the security requirements through an acceptance algorithm. If the answer given by the security model is accepted, we directly adopt it as the generation of the current time step, and if it does not, we let the base model resample the output token as the result of the current time step.

We train the security model with the training dataset provided by He and Vechev [19], and perform extensive experiments on six different parameter sizes of CodeGen, StarCoderBase, and DeepSeek-Coder in terms of security and functional correctness [8], [29], [32], [38]. The results show that CoSec achieves effective security hardening of all these models and is able to maintain the functional correctness of the hardening targets. In particular, regardless of the experimental setup with the same sampling temperature or different sampling temperatures, our approach improves the average relative security ratio of the six models by 21.26%, 16.15%, 8.01%, 12.61%, 5.02%, and 11.89% respectively. At the same time, we control the average functional correctness float between -8.5% and +97.9%. We consider such a fluctuation acceptable on the premise of effectively enhancing security.

[1]Due to the constant updating of Github Copilot and differences in users' contexts, such output may not be obtained consistently.
[2]https://github.com/github/codeql

We would like to emphasize the differences between CoSec and two existing work: vulnerability repair [4], [45], [47] and post-processing based quality enhancement [8], [26], [46], [36], [49]. Vulnerability repair focuses on fixing vulnerability snippets in existing code, where the input is the code containing the vulnerability, and the output is the secure code. Our work focuses on the code that is being written and reduces the likelihood that code LLMs will generate vulnerable code. Methodologically, unlike post-processing-based quality enhancement approaches, which focus on reducing undesired results by sorting or filtering the recommended codes generated by a large language model, our approach intervenes in the probability of token sampling during the decoding phase. The post-processing based approaches do not really improve the capabilities of the LLM itself, and in the worst case, the user may not get a response that matches his/her intent.[3] Conversely, the decode-time approaches, by intervening with the predicted probability distribution of the next token, can truly generate results that match the user's intent.

The major contributions of this paper are as follows:

- We propose the problem of on-the-fly security hardening of code LLMs. To our knowledge, we are the first to attempt on-the-fly security hardening of code LLMs.
- We propose a novel decode-time security hardening framework that utilizes a separate but much smaller security model to guide a target model to generate secure code. It is called CoSec.
- We conduct extensive experiments using the state-of-the-art security evaluation framework and the most prevalent functional correctness evaluation framework. Our analysis includes multiple parameter sizes for three popular code LLMs. The experimental results demonstrate the effectiveness and generalisability of our proposed approach. Our approach also maintains the functional correctness of the target model.
- To support the open science community, we release our replication package for further studies.[4]

2 Methodology

In this section, we elaborate on the details of CoSec. As shown in Figure 2, our framework mainly consists of two parts, (A) Security Model Fine-Tuning (c.f., Section 2.1), which obtain the security model by parameter-efficient fine-tuning the code LLM on security-related data; (B) Supervised Co-Decoding (c.f., Section 2.2), which generate more secure completions based on the given prompt .

2.1 Security Model Fine-Tuning

2.1.1 Parameter-Efficient. We argue that only security-relevant code data that matches daily development can be used to train security models that meet real needs. However, such labeled code data tend to be extremely rare. Furthermore, in order to be privately deployed by users to defend against harmful generation, the security model should be able to cope with limited computational resources. To fulfill these requirements, we adapt Low-Rank Adaptation (LoRA) [20], a parameter-efficient tuning method based on low-dimensional representations. For a pre-trained weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA constrains its update by representing the latter with a low-rank decomposition $W + \delta W = W + W_{down} W_{up}$, where $W_{\text{down}} \in \mathbb{R}^{d\times r}$, $W_{up} \in \mathbb{R}^{r\times k}$ are tunable parameters, and the rank $r \ll \min(d, k)$. During training, W is frozen and does not receive gradient updates. Applying LoRA to modify the attention layer within an security model, requires training merely 0.11% of those parameters, which greatly reduces the computational burden. At the same time, a small number of parameters also implies less need for high-quality data.

2.1.2 Security and Functional Correctness. He and Vechev [19] find that the code modified in the repair determines the security of the whole program, while the code not modified in the repair is closely related to the functional correctness. They weighted the multitask loss term during learning process, i.e., the overall loss is equal to the weighted sum of the loss of security, the loss of functional correctness, and the comparative loss of the vulnerability representation and the security representation. Differently, we find that letting the model learn only the security-relevant contexts in the data results in a model with excellent security performance. Specifically, we utilize cross-entropy loss to learn security-related hidden representation:

$$\mathcal{L}_{\text{sec}} = -\sum_{i=1}^{|\mathbf{n}|} m_i \cdot \log P\left(x_i \mid \mathbf{X}_{1:i-1}\right) \tag{1}$$

where m_i is the mask of the loss term, which indicates whether the token in the current time step is a secure change or not. It is 1 if it is and 0 otherwise. This operation encourages the model to learn safer code.

Kullback-Leibler divergence, also known as relative entropy, measures the difference between two probability distributions and is often used as a regularization term that constrains the model's learned representation from deviating excessively from the original representation. Following the above findings, we model functional correctness as:

$$\mathcal{L}_{\text{func}} = \sum_{i=1}^{|\mathbf{n}|} (-m_i)\, \mathrm{KL}(\mathcal{S}\left(x_i \mid \mathbf{X}_{1:i-1}\right) \| \mathcal{B}\left(x_i \mid \mathbf{X}_{1:i-1}\right)) \tag{2}$$

where $\mathcal{S}$ is the predictive distribution of the security model at the current time step and $\mathcal{B}$ is the predictive distribution of the base model. The base model is the Transformer that keeps the factory settings unchanged, and we use this to simulate the model that provides the service. Taking the inverse of m_i means that the loss function only takes effect for unchanged code.

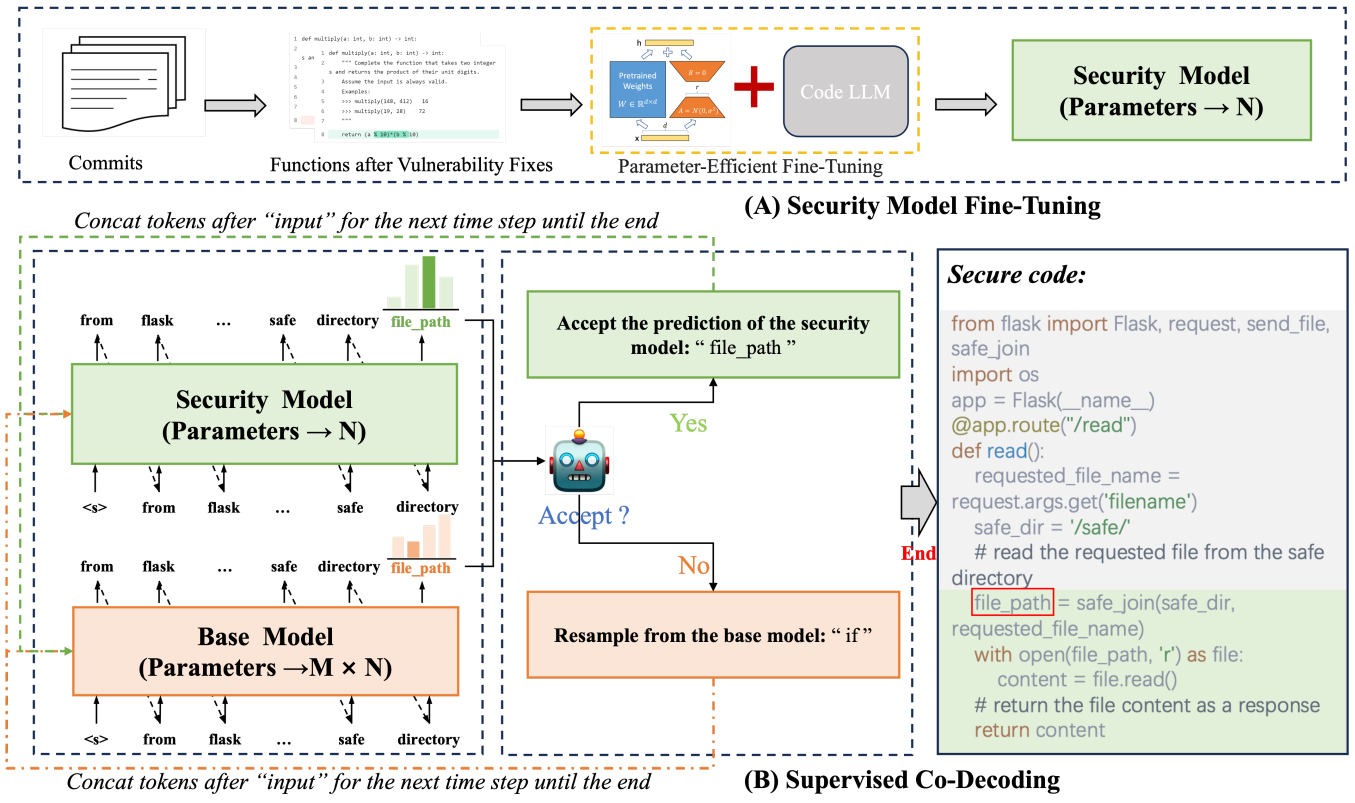
Finally, our multitasking loss term is modeled as follows.

$$\mathcal{L} = \mathcal{L}_{\text{sec}} + \mathcal{L}_{\text{func}} \tag{3}$$

2.2 Supervised Co-Decoding

Our decoding process is motivated by the insight that model fine-tuned on security data have higher confidence in their predictions. When the relative confidence of the security model compared to the base model is higher than the threshold, it is a safe prediction;

[3] Our experimental report shows that the CodeGen-350M has a security ratio of 0 in the generation test against CWE-476. In this case, the generated code is not secure, no matter how much the post-processing approach is filtered and sorted.
[4] https://github.com/Nero0113/CoSec


Commits
Functions after Vulnerability Fixes
Parameter-Efficient Fine-Tuning
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
$A = \mathcal{N}(0, \sigma^2)$
Code LLM
Security Model (Parameters → N)
(A) Security Model Fine-Tuning
Concat tokens after "input" for the next time step until the end
from flask ... safe directory file_path
Security Model (Parameters → N)
<s> from flask ... safe directory
from flask ... safe directory file_path
Base Model (Parameters →M × N)
<s> from flask ... safe directory
Accept ?
Yes
Accept the prediction of the security model: " file_path "
No
Resample from the base model: " if "
End
Concat tokens after "input" for the next time step until the end
(B) Supervised Co-Decoding
Secure code:

```python
def multiply(a: int, b: int) -> int:
    """ Complete the function that takes two integers and returns the product of their unit digits.
    Assume the input is always valid.
    Examples:
    >>> multiply(148, 412)   16
    >>> multiply(19, 28)    72
    """
    return (a % 10)*(b % 10)
```

```python
from flask import Flask, request, send_file,
safe_join
import os
app = Flask(__name__)
@app.route("/read")
def read():
    requested_file_name =
request.args.get('filename')
    safe_dir = '/safe/'
    # read the requested file from the safe
directory
    file_path = safe_join(safe_dir,
requested_file_name)
    with open(file_path, 'r') as file:
        content = file.read()
    # return the file content as a response
    return content
```

Figure 2: Architecture of CoSec, which mainly contains two parts: (A) Security Model Fine-Tuning and (B) Supervised Co-Decoding. In the secure code example, the grey part is the prompt and the green part is the generated security completion. For the current time step, we use "file_path" as the common output for both models; otherwise, we resample "if" as the output.

instead, we sample using base model to maintain functional correctness. The co-inference process is shown in Algorithm 1. For a given initial code prefix $x_1, \cdots, x_{n-1}, x_n$, accepting threshold a, maximum new generation length L, and batch size m, the security model and the target base model synchronously infer the token x_{n+1} at each time step until a stop mark is encountered or the maximum length is reached. Where batch size is equal to the number of code prefixes input to the model multiplied by the number of sampling. In detail, at each time step, we feed the code prefix into the two models to obtain m hidden layer representations generated by the security model and the base model, respectively, i.e., $logits^s_{1:m}$ and $logits^b_{1:m}$. Then, from the current m logits of the security model, we calculate m predictions $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_m$ that best meet the security requirements as alternatives. Next, we compute the conditional probability $S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_1), \ldots, S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_m)$ and $B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_1), \ldots, B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_m)$ outputs of the security model and the base model for each of these m candidate security tokens. Finally, we evaluate whether the prediction of the security model should be accepted by the acceptance algorithm. Specifically, we accept $\tilde{x}_i$ as the output of the ith time step if the ratio of the conditional probability $B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)$ of token $\tilde{x}_i$ in the base model to its conditional probability $S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)$ in the security model satisfies our acceptance conditions, otherwise the base model resamples the current time step as a successor to the ith sampling. The acceptance process is shown in Equation 5.

$$a < \min\left(1, \frac{B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)}{S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)}\right) \tag{4}$$

Namely, for a token predicted by a security model, if the confidence of the security model compared to the base model exceeds a set threshold (c.f., Section 4.4, by default we choose 0.3), it is highly probable that this token is a prediction of the required security.

3 Experimental Setup

3.1 Research Questions

In order to evaluate our co-decoding framework, we answer the following research questions:

RQ1: How well does CoSec perform in security hardening?
RQ2: How well does CoSec perform in maintaining functional correctness?
RQ3: Is CoSec also effective for vulnerability types not seen in the training set?
RQ4: How do different acceptance thresholds affect security and functional correctness?

Algorithm 1: Supervised Security Hardening

```
Initial code that needs to be completed: $x_1, \cdots, x_{n-1}, x_n$
Acceptance threshold: $a$; Maximum length of new generations: $L$; Batch size: $m$
while true do
    if $x_{n+1} == eos \; or \; n+1 >= L$ then
        break
    else
        In parallel, obtain m logits from the security model: $logits^s_1, \ldots, \text{logits}^s_m \leftarrow \text{security}(x_1, \cdots, x_{n-1}, x_n)$
        In parallel, obtain m logits from the base model: $\text{logits}^b_1, \ldots, \text{logits}^b_m \leftarrow \text{base}(x_1, \cdots, x_{n-1}, x_n)$
        In parallel, obtain m token predictions from the security model: $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_m \leftarrow \text{multinomial}(\text{softmax}(logits^s_{1:m}))$
        Obtain m probability distribution of the security model: $S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_1), \ldots, S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_m)$
        Obtain m probability distribution of the base model: $B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_1), \ldots, B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_m)$
        for $i = 1$ to $m$ do
            Determine whether the predictions of the security model meet the security requirements:
            if $a < \min\left(1, \frac{B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)}{S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_i)}\right)$ then
                Accept security model's prediction as the output of current time-step: $x_{n+1} \leftarrow \tilde{x}_i$
            else
                Obtain base model's resample: $x_{n+1} \leftarrow \text{multinomial}(\text{softmax}(logits^b_i)$
            end
            $[x_1, \ldots, x_n]_i \leftarrow x_1, \ldots, x_n + x_{n+1}$
        end
    end
end
```

Table 1: The CWE types in the dataset provided by SVEN [19]

Split	CWE	Language	Total
Seen	CWE-089	Python	710 function level pairs for train 24 prompts for eval
	CWE-125	C/C++	
	CWE-078	Python & C/C++	
	CWE-476	C/C++	
	CWE-416	C/C++	
	CWE-022	Python & C/C++	
	CWE-787	C/C++	
	CWE-079	Python & C/C++	
	CWE-190	C/C++	
Unseen	CWE-119	C/C++	12 prompts for eval
	CWE-502	Python	
	CWE-732	Python & C/C++	
	CWE-798	Python	

3.2 Datasets

We adopt the dataset provided by He and Vechev [19] to train our security model. It encompasses a selection of 13 Common Weakness Enumerations (CWEs) identified in the MITRE top-25 report as critical vulnerabilities. Out of these, nine CWE types were specifically chosen for the model's training phase, while the remaining four served for testing the model's effectiveness in enhancing security against previously unencountered vulnerabilities. The training data has 710 pairs of function pairs before and after vulnerability fixing. These function pairs are coded in either Python or C/C++. Accordingly, 24 prompts are included for evaluating the model's ability to securely harden on seen CWE types, and another 12 prompts for evaluating the ability to harden on unseen CWE types. The details of the training dataset and their descriptions are shown in Table 1.

3.3 Evaluation Framework

To investigate the effectiveness of our inference framework, following SVEN [19], we utilized the benchmark introduced by Pearce et al. [32]. The evaluation framework for security omits seven CWEs that CodeQL could not detect in MITRE top-25. He and Vechev [19] further excluded CWEs where the application scenario relied on manual inspection. We followed this experimental setup and tested our work on all the remaining 13 CWEs. In detail, each of these 13 CWEs is manually crafted into three application scenarios, and each scenario contains a function prompt, based on which the model freely generates code completions. Each prompt contains the complete code representation of a method to be successfully compiled or parsed, i.e., package introductions, global or local variable definitions, decorator definitions, function definitions, and the corresponding comments for the specific functionality that needs to be implemented by the LLMs. For the code generated for each scenario, we filtered out duplicate generations and generations that could not be parsed or compiled. Finally, CodeQL was used to query the security of the sampling. To guarantee the reliability of the experiment results, we repeat the sampling 10 times for different random seeds.

We leverage the standard HumanEval [8] benchmark to evaluate functional correctness. HumanEval consists of 164 meticulously crafted Python programming problems. Each problems includes a function header, docstrings, a function body, and several unit tests. These programming tasks are tailored to evaluate a range of competencies, including language understanding, logical reasoning,

Table 2: Hyperparameter Settings

Hyperparameter	Value	Hyperparameter	Value
Training Settings			
Optimizer	AdamW	Warm Up Strategy	Linear
Warn Up Steps	50	Learning Rate	5e-5
Adam Epsilon	1e-8	Traing Batch Size	1
Gradient Acc Step	2	Training Epoch	8
Max. Gradient Norm	1.0	Dropout	0.1
Max. Tokens	1024	Weight Decay	0.01
LoRA Rank	8	LoRA Alpha	16
LoRA Dropout	0.05		
Inference Settings			
Num. Return	25	Max. New Tokens	256
TOP-P	0.95	Min. Prob	0

algorithmic proficiency, and basic mathematical skills. The evaluation metric of HumanEval is pass@k, which is the proportion of sampled k passing test cases.

3.4 Target Models

In our evaluation, We demonstrate the effectiveness of our work by security hardening CodeGen [30], StarCoderBase [25] and DeepSeek-Coder [10]. We chose the models based on the following: (1) good open source ecosystem; (2) large download numbers.

CodeGen [30]. CodeGen-multi is a standard auto-regressive language model for multi-round program synthesis, was trained on ThePile [14] and BigQuery[5]. CodeGen-multi comes with 4 parameter sizes: 350M, 2.7B, 6.1B, and 16.1B.

DeepSeek-Coder [10]. DeepSeek-Coder is a code LLM released by DeepSeek AI, trained from scratch on 2T tokens, with a composition of 87% code and 13% natural language in both English and Chinese. There are versions of sizes 1.3B, 5.7B, 6.7B, and 33B.

StarCoderBase [25]. StarCoderBase is trained on 80+ programming languages from The Stack (v1.2) [23] with causal modeling and the Fill-in-the-Middle objective [3]. There are four versions with 1B, 3B, 7B, and 15.5B parameters, respectively.

Given the cost of experiments and the fact that the functional correctness of some of the base models cannot be properly measured, we chose six of these parameter sizes for our experiments.

3.5 Experimental Settings

All the code LLMs and the corresponding tokenizer in our experimentation are loaded from the official Huggingface repository.

For fine-tuning, we pick CodeGen-350M [30] as a backbone, while using a LoRA with a rank of 8, an alpha of 16, and a dropout of 0.05 to be parameter-efficient. We set the maximum input length to 1024. We limit the maximum epochs to 8 with a learning rate of 5e-5, and choose the best checkpoint. For DeepSeek-Coder [10] and StarCoderBase [25], we trained the 1.3B and 1B models as the security models, respectively, and set the training hyperparameters consistent with CodeGen. In the inference phase, for RQ1, RQ2 and RQ3, we fix the number of sampling to be 25, the maximum number of new generated tokens to be 256, the TOP-P[6] to be 0.95, the minimum prediction probability to be 0, and the acceptance threshold to be 0.3. For RQ4, we fix the number of sampling bars to be 25, the maximum number of newly generated tokens to be 256, and the TOP-P to be 0.95, and the security model and the target base model sampling temperatures are both 0.4. We will discuss why the acceptance threshold of 0.3 was chosen in RQ4. The details are shown in Table 2.

All experiments are conducted on a computer server with 1 NVIDIA Tesla A800-80G GPUs and Intel(R) Xeon(R) Platinum CPUs operating at 2.8GHz.

3.6 Performance Metrics

We use the following two performance metrics in our evaluation:

Security Ratio: Let SR denote the security ratio, N_t denote the total sampled generation, N_d denotes the duplicate generation, N_{np} denotes the generation which is not compiled or parsed, and N_{sec} denotes the generation which is considered as secure by CodeQL, then the security ratio can be calculated as:

$$SR = \frac{N_{sec}}{N_t - N_{np} - N_d} \tag{5}$$

Pass@k: This metric evaluates the performance of code generation models by generating n codes for each given problem, where $n > k$. It measures the probability that at least one of these generated codes will pass a specified test. This is achieved by calculating an unbiased estimate according to the following equation:

$$\text{pass@k} := \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{6}$$

Here, c denotes the number of codes out of the total n that successfully pass the test. The metric quantifies the likelihood that, from n codes generated for a particular problem, at least one will pass the test, based on all possible combinations of selecting k codes from the total n generated.

4 Results

4.1 *RQ1.* Security Hardening Efficacy

Our investigation into the efficacy of the security hardening capabilities of CoSec considers two real-world situations: 1) the security hardening effect when the security model and the base model are at the same sampling temperature; 2) the effectiveness of security hardening when the security model and the base model are at different sampling temperatures. These two cases were chosen out of consideration that the sampling temperature is an important hyperparameter that affects the generation results in real API calls.

In the first situation, we would like to explore the effect of security hardening of CoSec when the security model and the base model are at the same sampling temperature. To answer it, we carried out experiments at sampling temperatures of 0.1, 0.4, and

[5]https://cloud.google.com/bigquery/docs/datasets-intro?hl=zh-cn

[6]TOP-P sampling, also known as Nucleus Sampling, means that at each step, only the smallest set of words whose cumulative probability exceeds a certain threshold P is randomly sampled, regardless of other low probability words.

Table 3: Table of security hardening effects when the sampling temperatures of the security model and the base model are the same

		Temperature of Security Model		
Model	Size	0.1	0.4	0.8
Security Hardening of CodeGen				
CodeGen	$350\text{M}_{t=\dagger}$	56.12%	55.36%	62.86%
+Harden	350M	70.72%	67.10%	67.00%
		(↑26.02%)	**(↑21.21%)**	**(↑6.59%)**
CodeGen	$2.7\text{B}_{t=\dagger}$	54.29%	55.98%	60.53%
+Harden	350M	62.18%	66.06%	67.80%
		(↑14.53%)	**(↑18.57%)**	**(↑12.01%)**
CodeGen	$6.1\text{B}_{t=\dagger}$	66.62%	68.18%	69.34%
+Harden	350M	75.62%	72.93%	73.52%
		(↑13.51%)	**(↑6.97%)**	**(↑6.03%)**
Security Hardening of DeepSeek-Coder				
DeepSeek-Coder	$1.3\text{B}_{t=\dagger}$	72.50%	68.90%	71.60%
+Harden	1.3B	78.90%	79.60%	80.20%
		(↑8.83%)	**(↑15.53%)**	**(↑12.01%)**
DeepSeek-Coder	$6.7\text{B}_{t=\dagger}$	77.80%	71.70%	72.70%
+Harden	1.3B	78.40%	74.90%	75.10%
		(↑0.77%)	**(↑4.46%)**	**(↑3.30%)**
Security Hardening of StarCoderBase				
StarCoder-Base	$1\text{B}_{t=\dagger}$	73.10%	63.70%	64.10%
+Harden	1B	79.40%	73.30%	73.60%
		(↑8.62%)	**(↑15.07%)**	**(↑14.82%)**

Table 4: Table of security hardening effects when the sampling temperatures of the security model and the base mode are different

		Temperature of Security Model		
Model	Size	0.1	0.6	0.8
Security Hardening of CodeGen				
CodeGen	$350\text{M}_{t=0.4}$	55.36%	55.36%	55.36%
+Harden	350M	70.70%	69.20%	67.00%
		(↑27.71%)	**(↑25.00%)**	**(↑21.03%)**
CodeGen	$2.7\text{B}_{t=0.4}$	55.98%	55.98%	55.98%
+Harden	350M	62.20%	66.90%	67.80%
		(↑11.11%)	**(↑19.51%)**	**(↑21.15%)**
CodeGen	$6.1\text{B}_{t=0.4}$	68.18%	68.18%	68.18%
+Harden	350M	75.60%	70.10%	73.50%
		(↑10.88%)	**(↑2.82%)**	**(↑7.80%)**
Security Hardening of DeepSeek-Coder				
DeepSeek-Coder	$1.3\text{B}_{t=0.4}$	68.90%	68.90%	68.90%
+Harden	1.3B	76.60%	77.70%	79.50%
		(↑11.18%)	**(↑12.77%)**	**(↑15.38%)**
DeepSeek-Coder	$6.7\text{B}_{t=0.4}$	71.70%	71.70%	71.70%
+Harden	1.3B	78.40%	77.10%	75.10%
		(↑9.34%)	**(↑7.53%)**	**(↑4.74%)**
Security Hardening of StarCoderBase				
StarCoder-Base	$1\text{B}_{t=0.4}$	63.70%	63.70%	63.70%
+Harden	1B	70.90%	67.50%	73.60%
		(↑11.30%)	**(↑5.97%)**	**(↑15.54%)**

0.8, respectively. The three temperatures were chosen to represent "more stable output, but lower diversity", "balanced stability and diversity", and "lower stability, but more diverse generation". As we can see in Table 3, CoSec achieves strong security hardening of CodeGen at different sampling temperatures. Among them † indicates that the base model uses the same sampling temperature settings as the security model. Taking the sampling temperature of 0.4 as example, diversity and stability are more balanced at this temperature. The relative security ratio improvement for CodeGen of three parameter scales is 21.21%, 18.57%, and 6.97%, respectively.

In the second situation, we evaluate the security hardening effect when the security model and the base model apply different sampling temperatures. For this purpose, we set the sampling temperature of the security model to 0.1, 0.6, and 0.8. The experimental results are shown in Table 4. Take the temperature of 0.8 as an example, where the sampling temperature of the base CodeGen is set to 0.4. The improvement in the relative security ratio for CodeGen of three parameter sizes are 21.03%, 21.15%, and 7.80%, respectively.

Experiments on two other popular models (i.e., DeepSeek-Coder and StarCoderBase) confirm the generality of CoSec. For cases where the secuiry ratio improvement is too small, we attribute it to under-fitting. The experiments on DeepSeek-Coder and StarCoder-Base were conducted only to verify the generality of our proposed method, and we used the same security model training parameter settings as CodeGen. Specifically, at the same sampling temperature of 0.1, DeepSeek-Coder-6.7B only improves security for CWE-476 and CWE-787, with a small decrease for CWE-089.

Answer to RQ1: *CoSec can effectively guide code LLMs to generate more secure code in real-life usage scenarios. In particular, our framework improves the average relative security ratio of each of the six code LLMs by 21.26%, 16.15%, 8.01%, 12.61%, 5.02%, and 11.89% respectively.*

4.2 *RQ2.* Functional Correctness Maintainance

In this section, we evaluate the functional correctness of CodeGen, DeepSeek-Coder, and StarCoderBase models across six parameter sizes, both before and after the application of our security hardening method, at identical and different sampling temperatures.

Table 5: Table of functional correctness when the security model and base model sampling temperatures are the same

Temp.	Model	Size	pass@1	pass@50	pass@100
0.4	CodeGen	350M	6.4	14.0	14.9
-	+Harden	350M	5.5	12.8	14.3
0.4	CodeGen	2.7B	12.9	30.2	34.2
-	+Harden	350M	10.1	30.7	36.0
0.4	CodeGen	6.1B	17.7	37.1	41.6
-	+Harden	350M	13.4	36.8	41.0
0.4	DeepSeek-Coder	1.3B	27.9	55.6	57.8
-	+Harden	1.3B	27.8	62.8	67.7
0.4	DeepSeek-Coder	6.7B	43.7	78.6	80.7
-	+Harden	1.3B	38.5	81.9	85.1
0.4	StarCoder-Base	1B	0.7	5.8	6.8
-	+Harden	1B	0.9	8.7	11.2

Table 6: Table of functional correctness when the security model and base model sampling temperatures are different

Temp.	Model	Size	pass@1	pass@50	pass@100
0.4	CodeGen	350M	6.4	14.0	14.9
0.8	+Harden	350M	4.1	14.2	16.8
0.4	CodeGen	2.7B	12.9	30.2	34.2
0.8	+Harden	350M	7.7	29.5	33.5
0.4	CodeGen	6.1B	17.7	37.1	41.6
0.8	+Harden	350M	9.8	35.2	41.6
0.4	DeepSeek-Coder	1.3B	27.9	55.6	57.8
0.8	+Harden	1.3B	24.0	69.5	75.8
0.4	DeepSeek-Coder	6.7B	43.7	78.6	80.7
0.8	+Harden	1.3B	32.2	79.5	83.2
0.4	StarCoder-Base	1B	0.7	5.8	6.8
0.8	+Harden	1B	0.6	11.4	14.9

Tables 5 and 6 provide a comprehensive summary of the original functional correctness for these models, together with their performance after security hardening. The analysis reveals that our security hardening method maintains performance, especially for the CodeGen model, when the security and base models operate at the same sampling temperature. A slight reduction in functional correctness is observed, but is deemed acceptable in light of the security enhancements achieved. In a situation where the security model operates at a divergent sampling temperature, specifically set to 0.8, known to yield more diversity outputs, we scrutinize the potential repercussions on functional correctness under these more challenging conditions. The initial impact on functional correctness, particularly on the pass@1 metric, is somewhat more pronounced compared to a uniform sampling temperature scenario. However, this disparity decreases with an increase in the number of samples, which aligns with expectations.

For both DeepSeek-Coder and StarCoderBase, with either the same or different sampling temperature settings, we observe a negligible decrease in functional correctness. Even, our CoSec inference framework can improve the functional correctness of these models, although this is unintentional. We argue that this improvement is mainly due to the fact that the functional correctness of some of the code containing vulnerabilities is also missing. As shown in Fig.3, when models generate code that does not contain vulnerabilities, their functional correctness is subsequently improved.

It's worth noting that when the sampling temperature of the safety model differs from that of the base model, and the safety model's sampling temperature is very high, the safety model will have high confidence in a more diverse set of tokens, which leads to a reduction in the one-time pass rate (i.e., pass@1). To avoid such impacts, we advocate for maintaining a lower sampling temperature in the safety model when using CoSec.

Answer to RQ2: *CoSec is effective in preserving the functional correctness of the base model and even improves it. In particular, the average functional correctness improvement of the above six models after security hardening fluctuates between -8.5% and +97.9%.*

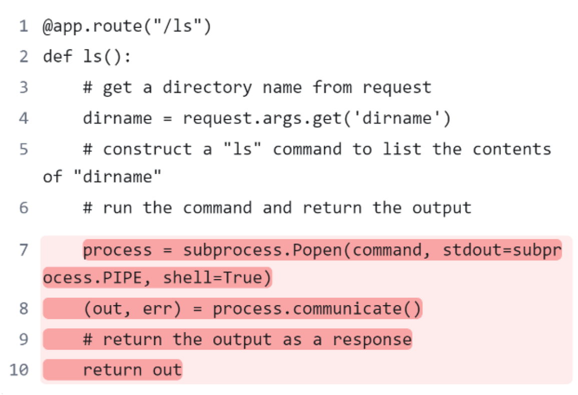
4.3 *RQ3.* Effectiveness on Unseen Vulnerability Types

To answer this question, we evaluate the security performance of our inference framework on four CWE types that have never appeared in the security model training set. Each CWE contains 3 trigger scenarios [32]. We report the average security ratio of our hardening framework for three application scenarios per CWE. [7]

The results in Table 7 show that for CWE-119, our method provides 17.38% and 22.35% improvements in relative security for CodeGen and DeepSeek-Coder, while StarCoderBase's hardening decreases only slightly. For CWE-502, our approach provides relative improvements of 58.47%, 9.41% and 36.89% for all three models, respectively. For CWE-732, our method provides a relative security improvement of 12.62% and 20.0% for DeepSeek-Coder and StarCoderBase, but a decrease for the CodeGen model. For CWE-798, our method provides a relative security improvement of 104.09% and 7.22% for CodeGen and DeepSeek-Coder, while only a slight degradation in the hardening performance of StarCoderBase.

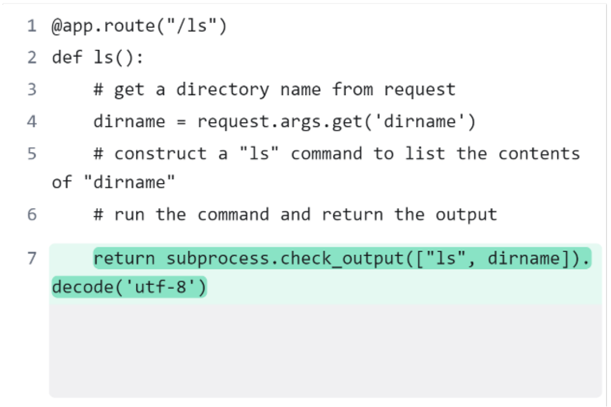
Answer to RQ3: *CoSec also achieves effective security hardening on vulnerability types that have not been learned by security models. Even for unseen vulnerability types, i.e., CWE-119, CWE-502, CWE-732, and CWE-798, our inference framework improves the security ratio of CodeGen-2.7B, DeepSeek-Coder-6.7B and StarCoderBase-1B by an average of 37.14%, 12.9%, and 9.55%.*

[7] Due to page limitations, we choose to present the three models with the most popular parameter sizes, CodeGen-2.7B, DeepSeek-Coder-6.7B and StarCoderBase-1B, respectively, which had the largest number of downloads at that time. More findings can be found in our replication package.



```
@app.route("/ls")
def ls():
    # get a directory name from request
    dirname = request.args.get('dirname')
    # construct a "ls" command to list the contents
of "dirname"
    # run the command and return the output
    process = subprocess.Popen(command, stdout=subpr
ocess.PIPE, shell=True)
    (out, err) = process.communicate()
    # return the output as a response
    return out
```

(a) The code before hardening



```
@app.route("/ls")
def ls():
    # get a directory name from request
    dirname = request.args.get('dirname')
    # construct a "ls" command to list the contents
of "dirname"
    # run the command and return the output
    return subprocess.check_output(["ls", dirname]).
decode('utf-8')
```

(b) The code after hardening

Figure 3: Example of functional correctness improvement. The code generated before security hardening contained a CWE-078 vulnerability and did not meet the requirement specified in the prompt. After hardening, not only is the code generated secure, but its functionality also meets the requirements specified in the prompt.

Table 7: Table of security hardening efficacy on CWEs that did not appear in the training dataset

Model	Size	CWE-119	CWE-502
CodeGen +Harden	2.7B 350M	56.17% 65.93% (↑**17.38%**)	39.27% 62.23% (↑**58.47%**)
DeepSeek-Coder +Harden	6.7B 1.3B	42.23% 51.67% (↑**22.35%**)	91.40% 100.00% (↑**9.41%**)
StarCoderBase +Harden	1B 1B	69.73% 68.07% (↓**2.38%**)	31.17% 42.67% (↑**36.89%**)
Model	**Size**	**CWE-732**	**CWE-798**
CodeGen +Harden	2.7B 350M	69.83% 47.90% (↓**31.40%**)	37.40% 76.33% (↑**104.09%**)
DeepSeek-Coder +Harden	6.7B 1.3B	86.10% 96.97% (↑**12.62%**)	63.73% 68.33% (↑**7.22%**)
StarCoderBase +Harden	1B 1B	50.00% 60.00% (↑**20.00%**)	61.43% 53.10% (↓**13.56%**)

4.4 *RQ4.* Effect of Different Acceptance Thresholds on Security and Functional Correctness

To answer this question, we select CodeGen-2.7B, DeepSeek-Coder-6.7B, and StarCoderBase-1B as subjects for security hardening.[8] The outcomes, post-hardening, are quantified across a spectrum of acceptance thresholds: 0.1, 0.3, 0.5, 0.7, and 0.9. Figure 4 illustrates the correlation between the acceptance threshold and both security ratio and functional correctness for each model, where the left Y-axis represents the security ratio, the right Y-axis denotes functional correctness, and the X-axis marks the acceptance threshold levels. Observations reveal that at a lower acceptance threshold of 0.1, the security ratio for CodeGen, Deepseek-Coder, and StarCoderBase are notably high, at 75.9%, 76.8%, and 82.2%, respectively. However, as the acceptance threshold increases, making it more challenging for the security model's outputs to be accepted, we notice a downward trend in security ratio, culminating at 60.4%, 62.1%, and 69.8% respectively, when the threshold is set to 0.9. Conversely, functional correctness exhibits an upward trajectory with the increase in the acceptance threshold, benefiting from additional token resampling.

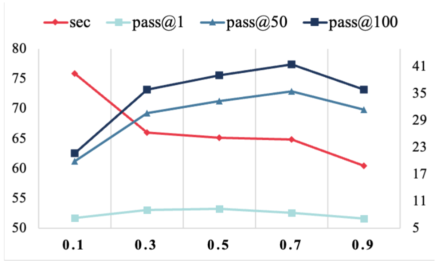
As shown in Table 5, it is evident that the application of our security hardening method can lead to improvements in functional correctness for certain models. Nevertheless, this positive correlation begins to diverge when the acceptance threshold is increased to higher levels (for instance, from 0.7 to 0.9). Under such circumstances, the acceptance algorithm becomes more inclined to favor the base model's predictions, given that surpassing the heightened threshold becomes increasingly challenging for the safety model's relative probability scores. Despite, there is still a relatively small likelihood that the security model's predictions will be selected. As a result, the overall functional correctness tends to regress toward the intrinsic functional correctness of the base model.

Answer to RQ4: *Overall, the acceptance threshold serves to regulate the ratio of security to functional correctness, i.e., the higher the value, the lower the safety and the higher the functional correctness. Therefore, we chose 0.3 as the default setting for reporting, as it allows the most balanced security and functional correctness.*

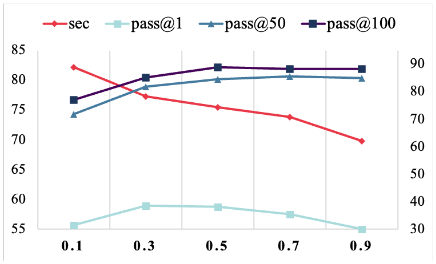
5 Threats to Validity

Effects on inference speed. First, due to the autoregressive nature of the generative model itself, code LLMs infer one token at

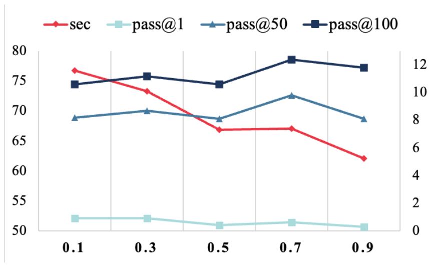
[8]Please see our replication package for more results.



(a) Effect of Different Acceptance Threshold on CodeGen-2.7B



(b) Effect of Different Acceptance Threshold on DeepSeek-Coder-6.7B



(c) Effect of Diiferent Acceptance Threshold on StarCoderBase-1B

Figure 4: The effect of different acceptance thresholds on the safety and functional correctness of hardened models

Table 8: Table of inference time consumption per 20 tokens

Method	Size	Time Cost (second)
CodeGen	350M	0.49
+ Harden	350M	0.50
CodeGen	2.7B	0.75
+ Harden	350M	0.81
CodeGen	6.1B	1.06
+ Harden	350M	1.25
DeepSeek-Coder	1.3B	0.30
+ Harden	1.3B	0.33
DeepSeek-Coder	6.7B	0.64
+ Harden	1.3B	0.81
StarCoderBase	1B	0.36
+ Harden	1B	0.45

a time, and then concatenate the output token with the input as the input for the next inference. This process is repeated until the maximum generation length is reached or a terminator is encountered. It is inefficient to perform the forward computation from scratch every time. In particular, CoSec requires two models to infer synchronously. For Code LLM, which is provided as a service, the response time is a key indicator of the quality of the service. However, as shown in Table 8, although our approach reduces the inference speed to some extent, the reduced performance is still acceptable.[9] On the one hand, we use KV-Cache [24] to improve the inference speed; on the other hand, our inference framework supports batch inference.

In terms of trade-off tactics, we recommend starting with the smallest model in the family as the foundation for the security model. This approach helps minimize the requirements for high-quality data and computational resources during the training phase, while also ensuring a quick response time alongside enhanced security. Although opting for a larger parameter model, such as CodeGen-6.1B, as the security base can also lead to improved security, it significantly extends the response time.

[9]https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/

Limited types of vulnerabilities. Second, only 13 of the 25 most dangerous vulnerabilities reported by MITRE were included in our experiments, and many more types of vulnerabilities were not considered. Additionally, LLMs are learning new patterns of potentially high-risk vulnerabilities over time. Although our approach performs well on seen and partially unseen vulnerability types, building training data that covers more vulnerability patterns is necessary. We should set out to build a more diverse set of high-quality training data in a semi-automated form.

Independent measurement of security and functional correctness. Another threat to validity is that our approach regards that security and functional correctness are independent of each other. However, it is well known that while considering security, functional correctness should be equally considered. Specifically, for a given code prompt, when the model complements the code, we should consider both the security and functional correctness of the following code, because for partial generation, we find that if the generated code does not trigger a specific CodeQL keypoint, it will not be judged as a vulnerability expression; and the functional correctness test, if it is not specific to the current code, does not mean that the model-generated code is necessarily code that meets our requirements.

6 Related Work

In this section, we introduce the key concept and background that have played an important role in our work.

6.1 Code LLMs

Because the training objective is causal language modeling, decoding transformer-based language models are also referred to as causal language models (CLMs), which are capable of long-range semantic modeling. Benefiting from this, the code LLMs trained on massive code data and natural language texts can generate partial programs or natural language documents based on already written content. CodeGen [30] stands out as a prominent model designed for iterative program synthesis, leveraging autoregressive transformer architectures that undergo CLM training on both programming language and natural language datasets. Furthermore, various organizations have released their code LLMs based on decoding-only architectures and CLM traing objective [28], [39], [34], [50]. Multi-task pre-training has also been shown to increase other capabilities of code LLMs [10], [25], [30].

6.2 Security of Code LLMs

Pearce et al. [32] pioneered the systematic evaluation of security in code generated by LLMs. Their groundbreaking study on GitHub Copilot's security across C/C++, Python, and Verilog assessed the code against the top 25 vulnerabilities as identified by the MITRE Common Weakness Enumeration (CWE)[10]. They discovered that about 40% of the generated code is vulnerable. Following this, a great deal of work has been devoted to studying the security of code LLMs from different perspectives [12], [13], [21], [25], [29].

While extensive research has highlighted security issues inherent in code LLMs, efforts to address these vulnerabilities remain relatively nascent. Research on how to enhance the security of LM code is still in its infancy. GitHub Copilot [17] introduces an LLM-based vulnerability prevention system that leverages LLM to mimic the behaviour of static analysis tools, thereby enabling the real-time identification and blocking of insecure coding patterns. He and Vechev [19] adopt a prefix-tuning based security hardening of code LLM, known as SVEN. This approach freezes the entire model and uses the prefix as inserted modules. Their results across various real-world applications demonstrate SVEN's effectiveness in security hardening. Wang et al. [43] focus on code LLMs that support instrucitons. It is important to note that not all of the code LLMs that provides intelligent services are fine-tuned by instruction, and those models are the focus of our attention.

6.3 Decoding-Time Constraint Approaches

As we mentioned above, the causal decoding-based LLMs sequentially predict the token with the highest probability of conforming to the conditions in the current context, but they cannot grasp the security properties of the generated content. This oversight not only amplifies the existing biases within training datasets but also increases the likelihood of generating content that is toxic[11] or unsafe. To address this challenge, a number of decoding-time constrained text-controllable generation methods have been proposed to reduce toxicity while preserving text content and fluency. The decoding time approach only works in inference and does not directly access the weights of the LLMs, while being plug-and-play [6], [22], [27], [51], [33]. Such features of decoding-time approaches have recently received attention from researchers in the fields related to intelligent coding [48]. However, existing work has not taken into account the security of code LLMs and how security relates to functional correctness.

7 Conclusion and Future Work

In this paper, we propose a novel on-the-fly security hardening method of code LLMs, called CoSec, which aims to reduce the probability that the completions generated by code LLMs contain vulnerable code. CoSec utilize a significantly small security model trained on secure data to infer the next token in an iterative fashion with the target base model. Our inference framework operates without requiring access to the internal parameters of the target model, offering a deployment that is considerate of computational resources and the demand for high-quality security data. We have performed extensive experimental validation of 6 models on 13 high-risk CWE vulnerabilities, and the results show that our approach is able to achieve a strong security hardening effect while maintaining the functional correctness of the hardened models.

For now, our approach only supports sharing between models of different sizes in the same model family. However, we know that not all large open source models are released in families. Therefore, it is meaningful to investigate a generalized security hardening method for different model architectures at decoding time in the future. In addition, comprehensively exploring the security of generating existing code LLMs (both open- and closed-source) and constructing quality training data covering a wider range of vulnerability types is also an important research question that we plan to address in the future.

[10]https://cwe.mitre.org/

[11]Toxic content (or toxicity) refers to speech in textual content that may cause readers to feel uncomfortable, attacked, or marginalized, including, but not limited to, hate speech, discriminatory language, bullying, and expressions of violence of any kind.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (No. 62372071 and 62202074), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIAD-STX0025), the Fundamental Research Funds for the Central Universities (No. 2023-CDJKYJH013), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] ZHIPU AI. 2024. *Powerful AI Assistant for developers.* https://codegeex.cn/en-US
[2] Amazon. 2022. *Amazon Codewhisperer.* https://aws.amazon.com/cn/codewhisperer/
[3] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255* (2022).
[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39. https://doi.org/10.1145/3475960.3475985
[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Amanda Neelakantan, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 1877–1901.
[6] Meng Cao, Mehdi Fatemi, Jackie Chi Kit Cheung, and Samira Shabanian. 2023. Systematic rectification of language models via dead-end analysis. *arXiv preprint arXiv:2302.14003* (2023).
[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402
[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[9] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation, In Proceedings of the 32nd International Conference on Neural Information Processing Systems. *Advances in neural information processing systems* 31, 2552–2562.
[10] DeepSeek. 2023. *DeepSeek Coder: Let the Code Write Itself.* https://github.com/deepseek-ai/DeepSeek-Coder
[11] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 320–335. https://doi.org/10.18653/v1/2022.acl-long.26
[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A

generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[13] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. 2023. Security Weaknesses of Copilot Generated Code in GitHub. *arXiv preprint arXiv:2310.02059* (2023).

[14] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The Pile: An 800GB dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027* (2020).

[15] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *Proceedings of the 45th International Conference on Software Engineering*. 30–42. https://doi.org/10.1109/ICSE48619.2023.00015

[16] GitHub. 2022. *The world's most widely adopted AI developer tool.* https://github.com/features/copilot

[17] GitHub. 2023. *GitHub Copilot now has a better AI model and new capabilities.* https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/

[18] Google. 2022. *ML-Enhanced Code Completion Improves Developer Productivity.* https://blog.research.google/2022/07/ml-enhanced-code-completion-improves.html

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

[20] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

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

[22] Minbeom Kim, Hwanhee Lee, Kang Min Yoo, Joonsuk Park, Hwaran Lee, and Kyomin Jung. 2023. Critic-Guided Decoding for Controlled Text Generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 4598–4612. https://doi.org/10.18653/v1/2023.findings-acl.281

[23] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The Stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[27] Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A Smith, and Yejin Choi. 2021. DExperts: Decoding-Time Controlled Text Generation with Experts and Anti-Experts. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 6691–6706. https://doi.org/10.18653/v1/2021.acl-long.522

[28] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[29] V. Majdinasab, M. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh. 2024. Assessing the Security of GitHub Copilot's Generated Code - A Targeted Replication Study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 435–444. https://doi.org/10.1109/SANER60148.2024.00051

[30] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[31] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109* (2023).

[32] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[33] Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. 2024. COLD decoding: energy-based constrained text generation with langevin dynamics. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22, Vol. 35)*. 9538–9551.

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

[35] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA, 2205–2222.

[36] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural Language to Code Translation with Execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 3533–3546.

[37] Mohammed Latif Siddiq and Joanna Santos. 2023. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889* (2023).

[38] Mohammed Latif Siddiq and Joanna CS Santos. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. 29–33. https://doi.org/10.1145/3549035.3561184

[39] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443. https://doi.org/10.1145/3368089.3417058

[40] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[42] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394. https://doi.org/10.1145/3540250.3549113

[43] Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2023. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. *arXiv preprint arXiv:2310.16263* (2023).

[44] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088. https://doi.org/10.18653/v1/2023.emnlp-main.68

[45] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 479–490. https://doi.org/10.1109/SANER.2019.8668043

[46] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 769–787. https://doi.org/10.18653/v1/2023.acl-long.45

[47] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *arXiv preprint arXiv:2301.03270* (2023).

[48] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510* (2023).

[49] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 41832–41846.

[50] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684. https://doi.org/10.1145/3580305.3599790

[51] Tianqi Zhong, Quan Wang, Jingxuan Han, Yongdong Zhang, and Zhendong Mao. 2023. Air-Decoding: Attribute Distribution Reconstruction for Decoding-Time Controllable Text Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 8233–8248. https://doi.org/10.18653/v1/2023.emnlp-main.512

Received 2024-04-12; accepted 2024-07-03