Improving Co-Decoding Based Security Hardening of Code LLMs Leveraging Knowledge Distillation

Dong Li, Shanfu Shu, Meng Yan, *Member, IEEE*, Zhongxin Liu, *Member, IEEE*, Chao Liu, Xiaohong Zhang, and David Lo, *Fellow, IEEE*

Abstract*—Large Language Models (LLMs) have been widely adopted by developers in software development. However, the massive pretraining code data is not rigorously filtered, allowing LLMs to learn unsafe coding patterns. Several prior studies have demonstrated that code LLMs tend to generate code with potential vulnerabilities. The widespread adoption of intelligent programming assistants poses a significant threat to the software development process. Existing approaches to mitigating this risk primarily involve constructing secure data that are free of vulnerabilities and then retraining or fine-tuning the models. However, such an effort is resource intensive and requires significant manual supervision. When the model parameters are too large (e.g., more than 1 billion) or multiple models with the same parameter scale have the same optimization needs (e.g., to avoid outputting vulnerable code), the above work will become unaffordable. To address this challenge, in previous work, we proposed CoSec, an approach to improve the security of code LLMs with different parameters by utilizing an independent and very small parametric security model as a decoding navigator. Despite CoSec's excellent performance, we found that there is still room for improving: 1) its ability to maintain the functional correctness of hardened targets, and 2) the security of the generated code. To address the above issues, we propose CoSec+, a hardening framework consisting of three phases: 1) Functional Correctness Alignment, which improves the functional correctness of the security base with knowledge disstillation; 2) Security Training, which yields an independent, but much smaller security model; and 3) Co-decoding, where the security model iteratively reasons about the next token along with the target model. Due to the higher confidence that a well-trained security model places in secure and correct tokens, it guides the target base model to generate more secure code, even as it improves the functional correctness of the target base model. We have conducted extensive experiments in several code LLMs (i.e., CodeGen, StarCoderBase, DeepSeekCoder and Qwen2.5-Coder), and the results show that our approach is effective in improving the functional correctness and security of the models. The evaluation results show that CoSec+ can deliver a 0.8% to 37.7% improvement in security across models of various parameter sizes and families; moreover, it preserves the functional correctness of the target base models—achieving functional-correctness gains of 0.7% to 51.1% for most of those models.

***Index Terms*—Large language models, code generation, software security, AI safety.**

Received 30 December 2024; revised 28 May 2025; accepted 17 July 2025. Date of publication 1 August 2025; date of current version 19 September 2025. Recommended for acceptance by M. Sabetzadeh. *(Dong Li and Shanfu Shu contributed equally to this work.) (Corresponding author: Meng Yan.)*

Dong Li, Shanfu Shu, Meng Yan, Chao Liu, and Xiaohong Zhang are with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China (e-mail: lidong@cqu.edu.cn; shanfu_shu@stu.cqu.edu.cn; mengy@cqu.edu.cn; liu.chao@cqu.edu.cn; xhongz@cqu.edu.cn).

Zhongxin Liu is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China (e-mail: liu_zx@zju.edu.cn).

David Lo is with the School of Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2025.3591791

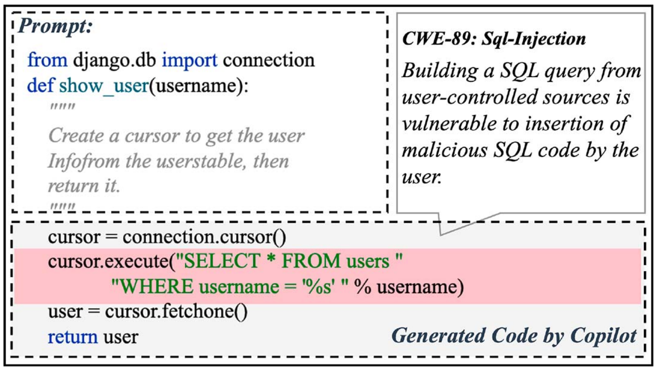
I. Introduction

RECENTLY, the rapid growth of computational power and the availability of massive datasets have driven significant breakthroughs in language modeling. Large language models (LLMs) equipped with billions of parameters—such as GPT-3 [1], LLaMA [2], and GLM [3]—now excel at a wide range of natural language tasks. By pre-training on a large amount of code and its associated natural language data, code LLMs (e.g., CodeGen [4], StarCoder [5], and DeepSeek-Coder [6]) can not only generate various code snippets that align with the programmer's intent but also perform a wide range of other software engineering tasks [7], [8], [9], [10], [11], [12]. Development assistance plug-ins (e.g., GitHub Copilot [13], AWS CodeWhisperer [14], CodeGeeX [15], etc.) based on code LLMs are serving a growing number of developers. GitHub's survey indicates that 92% of U.S. developers explicitly report using AI coding tools at work, with GitHub Copilot being the most popular [16].

The success of code LLMs relies on two main aspects: 1) massive open-source code and 2) causal language modeling. However, this inherent design also poses a risk for the practical application of code LLMs [5], [18], [19]. On the one hand, due to the crowdsourced nature of the pre-training dataset, it is not possible to completely filter out dangerous code from it; on the other hand, LLM suggests the most probable subsequent code guided by causal language modeling. Although the generated code is the most contextualized, it is unsafe. For example, when GitHub Copilot was prompted with the code in the upper left corner of Fig. 1, it generated the code in the gray box shown in the figure. In particular, the code snippet *"cursor.execute("SELECT ∗ FROM users WHERE username = '%s' " % username)"* exhibits a vulnerability to the CWE-89 weakness, as it incorporates a pattern susceptible to SQL injection attacks. In order to ascertain the generative safety of code LLMs, Pearce et al. [20] manually designed prompts for

0098-5589 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.


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

Fig. 1. Vulnerable code generated by Github copilot [17].

the application of GitHub Copilot in different programming scenarios and defined the concept of the security ratio of a code LLM (i.e., the proportion of generated code samples that are free of vulnerabilities). They then sampled the generated code, and the results showed that 40.73% of the code completions were insecure, yielding a security ratio of 59.27%. Extending this research, a large number of different empirical work has been carried out, and the findings show that these intelligent tools based on code LLMs will have a high chance of generating dangerous code [18], [5], [21], [22].

To alleviate the risk of generating vulnerable code from code LLMs, researchers set out to harden the output security of code LLMs. This involves making large language models more inclined to produce secure code that does not contain vulnerabilities during code generation (i.e., increasing the percentage of model-generated code that is secure). More importantly, any approach adopted for this purpose must not compromise functional correctness of the LLMs(i.e. the model's capability to generate code that both compiles successfully and aligns with the developer's intent.), thereby preserving its overall utility. He and Vechev [23] proposed the first SOTA security hardening method of code LLMs, called SVEN, which utilises the attention mechanism to efficiently change the computation of the hidden state by applying a pair of opposing prefixes to learn the representation of vulnerability and security, respectively. Such an operation guides the iteration of code LLMs towards security-compliant generation.

Although SVEN has demonstrated outstanding performance in enhancing the generative security of code LLMs, modern models are increasingly deployed as services (i.e., Model-as-a-Service). In such scenarios, direct fine-tuning or parameter-efficient fine-tuning approaches face the following limitations: **1) require access to internal parameters; 2)require repeated training; 3) lack of deployment flexibility.**

To cope with the above limitations, we proposed a co-decoding approach called CoSec in our previous work, to respond to the security requirements of code LLMs in model-as-a-service (MAAS) scenarios [24]. CoSec deploys a small code language model trained on a security-related dataset to infer the next token simultaneously with the target code LLMs, continuing until either the maximum length is reached or a terminator is encountered. The acceptance algorithm in this process is responsible for determining whether the next token comes from the security model or the target model by the relative confidence of these two models. Because security hardening is only performed during inference, CoSec does not need to access the internal parameters of the hardened model. Second, CoSec can harden models of any size in the same model family, avoiding repeated training. Finally, CoSec can be deployed only on the service links that require security hardening without affecting other service links. It is plug-and-play and does not require downtime.

Despite its outstanding performance, there is still room for improvement in CoSec. To gain a deeper understanding of the limits of CoSec, we performed further analysis within the testing framework proposed by Pearce et al. [20]. According to the empirical results, we have the following findings: **1) CoSec could be improved in functional correctness.** Although CoSec improves functional correctness under multiple sampling conditions (e.g., Pass@50), it becomes unstable in fewer sampling scenarios (e.g., Pass@1) when a smaller security model (350M) is used to harden a larger base model (6B). The root cause lies in the functional correctness limitations of the smaller model, which, without targeted fine-tuning, struggles to reliably align its outputs with the requirements of the larger model. **2) CoSec could be improved in security hardening ability.** When training is insufficient, the security model lacks the confidence to counteract the baseline's security tendencies for sparsely represented CWE types. For example, if the security model has a confidence level of 0.85 for `file_path`, but the target model is 0.90, the secure token suggestion from the security model would be overridden during co-decoding. Consequently, even though the security model alone achieves 85.0% security, its hardening only increases the security ratio of the 350M base model to 67.1%.

For better performance, we present our optimized training and inference pipeline, **CoSec+**. Specifically, CoSec+ consists of three stages: **(1) Functional correctness alignment.** We apply knowledge distillation techniques to align the baseline of the security model with a more powerful 'teacher' model, thus improving its functional correctness. **(2) Security training.** We optimize the training objectives and refine the data distribution. This ensures that the model has sufficient opportunities to learn the target secure coding patterns. It also minimizes the negative effects of inconsistent training objectives. Together, these improvements improve the overall security hardening capability. **(3) Co-decoding.** The security model and the target model work together to decode the secure code.

In order to reliably align the predictions of a security model with the requirements of a target model, an intuitive approach is to use knowledge distillation to transfer the generative patterns of a larger model into the base of security model. However, as a basis for knowledge distillation, Kullback-Leibler Divergence (KLD) is often used for finite prediction space tasks like discriminative modeling. For open-ended text generation tasks, which is usually the case for LLMs, the output space is much more complex. Minimizing the traditional KLD leads the student model to assign unreasonably high probabilities to the low-probability regions in the teacher's distribution, thus

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

producing very unlikely samples in autonomous generation [25]. Conversely, the reverse Kullback-Leibler Divergence (rKLD) focuses more on the regions where the teacher model is most confident. At the same time, to enable the student model more swiftly and directly concentrate on the teacher's optimal predictions and make more decisive choices for the correct answers than the teacher, we employ standard cross-entropy. This approach encourages the student to strike an optimal balance between hard labels (i.e., the ground truth of the next token) and soft labels (i.e., the probability distribution of the next token predicted by the teacher model) [26].

The types of vulnerabilities in the training corpus and the imbalance of programming languages are the main reasons why CoSec's security model does not provide a level of confidence sufficient to change the direction of security. We use direct oversampling in security training to give sufficient exposure to a few classes. We describe the oversampling approach in detail in Section IV. Experiments show that such an approach is simple and effective. In addition, during security training, we find that Adoption of losses inconsistent with the alignment of the functional correctness phase weakens the high certainty learned from knowledge distillation, which reduces Usability of generated code. For this reason, we replaced the weighted KLD with the weighted rKLD to obtain better maintenance of functional correctness.

We conducted a comprehensive evaluation of the security hardening effect of CoSec+ on the CodeGen, StarCoderBase, DeepSeek-Coder, and Qwen2.5-Coder model series, as well as its impact on the functional correctness of the target models. The experimental results show that CoSec+ improves security by 12.3% to 37.7% across three model families. Notably, we achieved a 0.8% security improvement for a 14B target model using a 1.5B security model. In terms of functional correctness, CoSec+ provides an improvement of 1.6% to 82.8% for most target models.

In summary, our contributions are as follows:

- We have conducted an in-depth investigation of CoSec to provide insights into its limitations and what are areas of improvements.
- We propose CoSec+, a novel three-stage optimization pipeline built on knowledge distillation, to address the shortcomings of CoSec. The Functional Correctness Alignment phase leverages advanced teacher models and rKLD to establish a foundation of security model with excellent functional correctness, optimized security training to provide higher security and functional correctness for the security model, and finally co-decoding to achieve security hardening of the target model.
- We performed extensive experiments with CoSec+ to evaluate its effectiveness in terms of security hardening and maintenance of functional correctness. The results of the experiments proved the effectiveness of CoSec+.
- We publicly release the CoSec+ reproduction package, including the dataset, three distilled base models, and the security training code[1].

[1]https://github.com/Nero0113/CoSec-plus

As an extended work of our previous work [24], which proposed and evaluated CoSec, this paper significantly extended our previous work in the following ways:

- We conducted an empirical study to gain insight into the shortcomings of CoSec (a black-box approach to harden the security of code LLMs) in functional correctness maintenance and security enhancement.
- To further improve the black-box security hardening approach in security enhancement and functional correctness maintenance, we propose a novel optimization pipeline to improve CoSec through knowledge distillation, which is referred to as CoSec+.
- In addition to the six models mentioned in previous work, we have added security hardening experiments on the StarCoderBase-7B and the Qwen2.5-Coder-14B.
- We conducted a more comprehensive evaluation of the proposed training pipeline. In this work, we comprehensively evaluate both metrics and add evaluation of vulnerability type generalization and impact of different components on the inference framework. The experimental results show the effectiveness of the proposed method.

II. Background and Related Work

In this section, we discuss the foundational concepts and important background related to CoSec+, including: security of Code LLMs [4], [5], [6], [27], [28], [18], knowledge distillation for LLMs [25], [29], [30], and decoding-time constraints approaches [31], [32], [33].

A. Security of Code LLMs

Code generation with LLMs. Causal language models (CLMs), trained on large-scale code and text in a self-supervised manner, excel in zero-shot generalization and semantic understanding. Built on Transformer decoders, CLMs generate code by predicting the next token from prior code or natural language. [4], [5], [6] CodeGen is the pioneering open-source code LLM trained using a decoder-only Transformer architecture on six programming languages alongside natural language, which excels particularly in program synthesis tasks [4]. In addition to this, other code LLMs have been released in succession [34], [27], [5]. In order to cope with the real scenario coding requirements, some other pre-training tasks (e.g., Fill-in-the-middle, FIM) are introduced to enhance the code's large model capability [6], [35], [36].

Generation Risks of Code LLMs. As intelligent plug-ins powered by code LLMs gain traction, their generative risks are becoming increasingly evident. These risks stem from both the scale and quality of pre-training data and the nature of causal language models (CLMs). Due to the crowd-sourced origin and sheer volume of training data, harmful code and undiscovered vulnerabilities cannot be fully eliminated. Moreover, CLMs may unintentionally generate insecure code when prompts align with learned vulnerability patterns, posing significant threats to software security. Therefore, to investigate the security of smart coding assistants, Pearce et al. [20] conducted the first survey of the 25 most dangerous vulnerability types that GitHub Copilot

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

may encounter in actual C/C++ and Python programming scenarios. The results show that more than 40% of the generated code is insecure. Muhammad et al. [17] introduced SALM, a framework that can comprehensively evaluate the security of LLM generation from both static and dynamic perspectives. Experiments have found that although GPT-4 is the model with the highest functional correctness, the code it generates has a large number of vulnerabilities. Several other empirical works have confirmed the generation risk of open-source and closed-source code LLMs from multiple perspectives, respectively [28], [18], [5].

Security Hardening for Code LLMs. While many studies highlight the security risks of code LLMs, few propose effective solutions. Research on enhancing their security remains nascent. GitHub Copilot [37] uses an LLM-based system to mimic static analysis tools and block insecure patterns (e.g., hardcoded credentials, SQL/path injections) in real time. Wang et al. [38] improve security by embedding vulnerability hints into prompts for instruction-tuned models. He and Vechev [23] adopt a security hardening approach based on prefix tuning for the code LLMs, known as SVEN. Experimental results show that SVEN is effective in hardening code LLMs in a variety of real-world programming scenarios. Despite the very good hardening results achieved by SVEN, the white-box approach still has some limitations in real MAAS applications, i.e., (1) requires access to internal parameters; (2) cannot be shared by models with different parameter sizes; (3) cannot adjust the security reinforcement strength. To cope with these limitations, in our previous work, we proposed CoSec [24], a security hardening method based on collaborative decoding of security models. This black-box approach guides the generation of more secure code by not requiring access to the internal parameters of the target model. While CoSec copes well with the above limitations, there is still a lot of room for improvement in terms of security and functional correctness.

B. Knowledge Distillation for LLMs

Knowledge distillation (KD) reduces model complexity by training a smaller student model to mimic a larger teacher model's output distributions, enabling comparable performance with lower computational cost. However, previous KD methods have mainly been applied to white-box classification models [39], [40] or fine-tuning small models to mimic black-box model outputs such as ChatGPT [41], [42]. With the popularity of open-source LLMs, it has become important to effectively align small models with large models. Gu et al. [25] first extended KD to autoregressive LLM when they replaced the Kullback-Leibler Divergence (KLD) dispersion goal in the standard knowledge distillation approach with reverse Kullback-Leibler Divergence (rKLD) to help prevent overestimation of the distribution of student models for teachers in low-probability regions. Wen et al. [43] proposed a series of sequence-level distillation methods, called f-DISTILL, whose symmetric losses outperform asymmetric losses by avoiding extreme mode averaging and collapsing. In order to improve the efficiency and effectiveness of knowledge distillation for autoregressive language models, Ko et al. [44] proposed DISTILLM, which consists of two main components 1) a skew Kullback-Leibler divergence loss, and 2) an adaptive off-policy approach.

C. Inference-Time Optimization Methods

Leveraging collaborative inference between large and small models for more flexible and energy-efficient performance optimization has now become a hot topic in both academia and industry. Leviathan et al. [45] proposed speculative sampling, which uses lightweight models (draft models) to quickly generate candidate token sequences, and then the target large model verifies their plausibility in parallel, thereby achieving an inference acceleration effect of 2–4 times. This algorithm has now been incorporated into the Transformer function library.[2]. Liu et al. [31] presented DEXPERTS, a method for predicting reweighting based on a safety model and a toxicity model. Experiments show that this method can effectively guide LLMs to the output that meets the expectation, while maintaining the fluency and diversity. Kim et al. [46] proposed a method for controlled text generation called CriticControl, which uses reinforcement learning and weighted decoding. This method uses a "critic" trained from a reward model to weight the output distribution. In the industry, companies such as Apple are also using edge-cloud collaboration to enhance intelligence in the next generation of Apple intelligence[3]. Such an optimization paradigm has recently attracted the attention of researchers in related fields of intelligent coding [47]. However, existing work has not yet considered the security of code LLMs.

III. A Systematic Evaluation of CoSec

To the best of our knowledge, CoSec [24] is currently the first decode-time constrained security hardening method of code LLMs. Therefore, for the sake of ensuring the coherence of the article, we first need to understand how it works, then analyze its impact, identify its issues and underlying causes, and ultimately explore ways to improve it. In particular, this section discusses the following research questions:

- **RQ1:** *What affect CoSec's effectiveness of maintaining functional correctness?* We analyze the differences in functional correctness between the security model, the target model, and CoSec, and summarize the reasons affecting functional correctness. This analysis serves as a foundation for inspiring improvements in the functional correctness of CoSec.
- **RQ2:** *What affect CoSec's effectiveness of security hardening?* We explored the types of vulnerabilities CoSec does not perform well enough, analyze the percentage of vulnerabilities in the training data, and summarize the factors that influence the effectiveness of security hardening. This analysis provides insights for enhancing security.

[2]https://huggingface.co/docs/transformers/en/index
[3]https://machinelearning.apple.com/research/introducing-apple-foundation-models

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

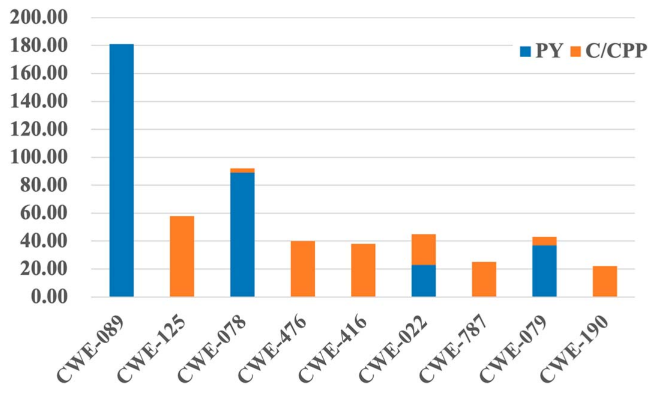


Fig. 2. The data radio in training set.

A. Dataset

In the previous work, we utilized the dataset curated by He and Vechev [23] for training the security model of CoSec. The training set contains the 9 critical vulnerabilities highlighted in the MITRE Top 25 report, and each vulnerability contains the corresponding function pair before and after the fix, as well as the edit points for the corresponding modifications. The dataset is primarily oriented towards Python and C/CPP. The percentage of each CWE type as well as programming language in the dataset is shown in Fig. 2.

B. Evaluation Frameworks

In this paper, two publicly available evaluation frameworks are used. We briefly introduce these two frameworks below.

1)Functional Correctness. We use the standard Humaneval benchmark to evaluate the impact of the proposed method on the functional correctness of the target base models (whether positively or negatively). [48]. HumanEval provides a function signature, docstring, function body, and unit tests for each task as input. The model is required to output code in the expected format, and the functional correctness is determined by measuring the proportion of test cases passed. HumanEval was designed with the goal of simulating real-world programming tasks and is therefore more comprehensive and rigorous than traditional synthetic datasets.

The core evaluation metric of HumanEval is **Pass@k**, which measures the probability that a model will correctly solve a task when generating multiple code samples. Specifically, code LLMs generate n codes for each given problem, in which HumanEval samples k times, $n >> k$. To avoid bias between the quality of model generation and the number of samples, HumanEval estimates Pass@k using a statistical formula:

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

2)Security Rate. We use the benchmark proposed by Pearce et al. [20] to evaluate security rates, whose objective is to evaluate the security of intelligent assistants in real programming scenarios. This framework omits 7 CWEs that CodeQL could not analyze in MITRE top-25. He and Vechev [23] further exclude the types of CWEs for which the application scenario relies on manual inspection. We followed this experimental setup and tested our work on the remaining 13 CWEs. Specifically, each of these 13 CWEs was manually crafted into three different application scenarios, each of which contained a function hint on which code was freely generated based on the hint model. Each hint contains the code components necessary for successful compilation or parsing, i.e., package introductions, global or local variable definitions, decorators, function definitions, and corresponding annotations for the specific functionality that LLM needs to implement. For the code generated for each scenario, we filter out generation that could not be parsed or compiled as well as duplicate generation. Finally, CodeQL was used to query the security of the sampling. To ensure the reliability of the experimental results, we repeated the sampling 10 times for different random seeds. The security ratio is calculated as follows:

$$Security\_Ratio = \frac{N_{sec}}{N_t - N_{np} - N_d} \tag{2}$$

Where, N_t, N_{sec}, N_{np}, and N_d denote the total number of code snippets generated for the same prompt, the number of de-duplicated code snippets analyzed as safe by CodeQL, the number of code snippets that fail to compile, and the number of duplicates in the generated code, respectively.

C. A Brief Review of CoSec

As the first decoding-time constrained security hardening method for code LLMs, CoSec consists of two main parts: 1) Security Model Fine-tuning. This part selects the model with the smallest number of parameters in a certain family as a base (e.g., CodeGen-350M). Then, the function fragments and change information when fixing vulnerabilities are used as input, and the weighted cross-entropy and KLD are utilized as the combined multitasking loss function to obtain the security model. 2) Supervised Co-Decoding. We utilize the security model to reason collaboratively with the target model that is being secured by the security hardening in a designed decoding framework. Specifically, for a given prompt, both models give their predictions simultaneously. When the confidence level of the token predicted by the security model for the current time step is higher than that of the target model, we take it as the common output of the two models, otherwise the target model resamples it as the output. This process is iterated until the maximum length or termination.

D. RQ1: What Affect CoSec's Effectiveness of Maintaining Functional Correctness?

In previous work [24], we have found that if a model with a smaller parameter size is chosen as a base for secure training, there is then a gap between its own functional correctness and that of the larger model, and this gap increases further as the parameter size increases. Therefore, in RQ1, we compare the difference in functional correctness between the chosen safety model and itself before and after safety fine-tuning, as well

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

as with the target model. We further analyze CoSec's failure to pass the HumanEval example during co-decoding to gain insight into the reasons for the failure. We conducted experiments on CodeGen, Deepseek-Coder, and StarCoderBase[4], respectively. Since CoSec is a cross-parameter security hardening method within a model family, it is essential to ensure that the issue arises from CoSec's own limitations rather than a coincidental degradation caused by differences between models. We only consider the cases of Pass@1, Pass@5, and Pass@10. In actual programming, the real-time response of the model needs to be considered.

As reported in Table I, we can see that the functional correctness of CodeGen-350M decreases after being trained on security, with an average decrease of 11.0%. Therefore, in its own security hardening, it has a negative impact on the functional correctness of the target model of an average of 12.2%. This negative impact further increases with the increase in the number of participating targets of the hardening. Co-inference decreases the functional correctness of CodeGen-2.7B by an average of 19.1%. For the hardening target 6B, the functional correctness of the security model is much lower than that of CodeGen-6B itself. In this case, the Pass@1, Pass@5 and Pass@10 scores of CodeGen-6B decreased by 24.3%, 15.0% and 7.3% respectively. For StarCoderBase, our safety training improved the Pass@1, Pass@5, and Pass@10 scores of the 1B model by 1.0%, 2.1%, and 4.6%, respectively. However, this improvement was insufficient to significantly impact the functional correctness of the 7B target model, which still saw an average decrease of 11.2%. Similarly, although the security model of Deepseek-Coder 1.3B has improved its functional correctness after security training, it is still far from the functional correctness of the target model 6.7B. Therefore, it also reduces the generation ability of the target model by an average of 10.5%.

Answer to RQ1: ① Directly training a security model will affect its functional correctness. Under the same training hyperparameters, a model with larger parameters has better training stability and can maintain its own functional correctness. ② The functional correctness after the safety reinforcement depends on the functional correctness of the safety model itself. The greater the gap between the target function and its correctness, the greater the impact. ③ As the number of samples increases, the impact of the safety model on the functional correctness of the target model is diluted. However, more samples will inevitably increase the response time.

TABLE I
IMPACT OF COSEC ON FUNCTIONAL CORRECTNESS

Model	Type	HumanEval		
		Pass@1	Pass@5	Pass@10
Functional Correctness of the CodeGen series				
CodeGen -350M	Origin	6.4	9.5	10.9
	Security Model*	5.5	8.5	10.0
		(↓**14.1%**)	(↓**10.5%**)	(↓**8.3%**)
CodeGen -2.7B	Origin	12.9	20.8	23.6
	+CoSec	10.1	16.4	20.2
		(↓**21.7%**)	(↓**21.2%**)	(↓**14.4%**)
CodeGen -6.1B	Origin	17.7	24.6	27.4
	+CoSec	13.4	20.9	25.4
		(↓**24.3%**)	(↓**15.0%**)	(↓**7.3%**)
Functional Correctness of the StarCoderBase Series				
StarCoder Base-1B	Origin	13.3	19.9	22.9
	Security Model*	13.1	19.4	22.1
		(↓**1.5%**)	(↓**2.5%**)	(↓**3.5%**)
StarCoder Base-7B	Origin	20.0	31.6	36.0
	+CoSec	17.3	27.8	33.1
		(↓**13.5%**)	(↓**12.0%**)	(↓**8.1%**)
Functional Correctness of the Deepseek-Coder Series				
DeepSeek-Coder-1.3B	Origin	27.9	42.0	46.9
	Security Model*	27.6	43.5	49.4
		(↓**1.1%**)	(↑**3.6%**)	(↑**5.3%**)
DeepSeek-Coder-6.7B	Origin	43.7	63.0	69.2
	+CoSec	38.5	60.3	67.1
		(↓**24.3%**)	(↓**4.3%**)	(↓**3.0%**)

*Refers to the trained security model based on the origin model in CoSec.

E. RQ2: What Affect CoSec's Effectiveness of Security Hardening?

As seen from Table II, even after applying CoSec, there remains a significant gap in security compared to security models. In detail, for CodeGen-2.7B and CodeGen-6.1B, there is a 22.2% and 14.2% gap in the overall security compared to the security model, respectively. The largest gaps are found for CWE-078 and CW-476, which amount to 65.4%, 42.5%, 49.5%, and 54.3% gaps, respectively. Similarly, on StarCoderBase and DeepSeek-Coder that there is still an 8.5% to 14.3% gap after hardening relative to the security model. After the security reinforcement, different effects are observed for different CWE types, e.g., for CodeGen, the security reinforcement on CWE-125 exceeded the security model. For StarCoderBase and DeepSeek-Coder on the other hand, it surpassed the security model on CWE-022. We argue that this gap is caused by the uneven distribution of training data. Because, security models do not learn enough on long-tailed data, the confidence level is not sufficient to change the direction of the prediction when providing predictions for that type. In particular, as can be seen from the distribution of the training data in Fig. 2, there are two reasons for the imbalance of the data, on the one hand, in terms of the total amount of data, Python related data is much more than C/CPP; on the other hand, in the same vulnerability type, C/CPP accounted for a much smaller proportion than Python.

[4]We inserted the prefix tag <fim_prefix> in front of the prompt based on StarCoderBase's official use case and updated CoSec's effectiveness on StarCoderBase series.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

TABLE II
IMPACT OF COSEC ON SECURITY

Model	Type	Security Radio (CWE)									Total
		- 089 -	- 125 -	- 078 -	- 476 -	- 416 -	- 022 -	- 787 -	- 079 -	- 190 -	
Security of the CodeGen series											
CodeGen -350M	Security Model*	100.0%	60.2%	89.7%	63.7%	63.7%	100.0%	87.7%	100.0%	100.0%	85.0%
CodeGen -2.7B	+CoSec	66.5%	72.9%	52.8%	32.2%	82.6%	87.7%	56.5%	65.4%	78.0%	66.1%
		↓33.5%	**↑21.1%**	**↓65.4%**	**↓49.5%**	**↑29.7%**	**↓12.3%**	**↓35.6%**	**↓34.6%**	**↓22.0%**	**↓22.2%**
CodeGen -6.1B	+CoSec	75.0%	64.7%	51.6%	29.1%	83.3%	100.0%	65.1%	96.9%	90.9%	72.9%
		↓25.0%	**↑7.5%**	**↓42.5%**	**↓54.3%**	**↑30.8%**	**-**	**↓25.8%**	**↓3.1%**	**↓9.1%**	**↓14.2%**
Security of the StarCoderBase series											
StarCoder Base-1B	Security Model*	100.0%	75.0%	100.0%	43.2%	93.6%	42.9%	97.9%	100.0%	65.4%	79.8%
StarCoder Base-7B	+CoSec	89.6%	72.8%	54.0%	23.8%	100.0%	51.5%	97.6%	100.0%	68.0%	73.0%
		↓10.4%	**↓2.9%**	**↓46.0%**	**↓44.9%**	**↑6.8%**	**↑20.0%**	**↓0.3%**	**-**	**↑4.0%**	**↓8.5%**
Security of the DeepSeek-Coder series											
DeepSeek-Coder-1.3B	Security Model*	100.0%	100.0%	100.0%	50.5%	100.0%	92.3%	64.0%	100.0%	80.0%	87.4%
DeepSeek-Coder-6.7B	+CoSec	96.0%	77.1%	36.0%	2.3%	100.0%	95.0%	74.0%	100.0%	94.0%	74.9%
		↓4.0%	**↓22.9%**	**↓64.0%**	**↓95.4%**	**-**	**↑2.9%**	**↑15.6%**	**-**	**↓17.5%**	**↓14.3%**

*Refers to the trained security model based on the origin model in CoSec.

Answer to RQ2: ① At the data level, there is an imbalance in the training data of the security model in two ways: an imbalance in the total number of different programming languages for different CWEs, and 2) an imbalance in the percentage of different programming languages for the same CWEs. ② At the model level, there is a large gap between the hardened security and the security model due to the lack of confidence learned by the model on a few categories.

IV. APPROACH

Our empirical study shows that CoSec is effective in generating security for models with different number of parameters in the same model family. However, the overall security and functional correctness of CoSec could be further improved. To address this issue, we propose **CoSec+**. Fig. 1 illustrates the overall framework of CoSec+, which consists of three phases: 1) Functional correctness alignment. Functional correctness of the base model of the security model is aligned to the best model in the family through knowledge distillation. 2) Safety fine-tuning. Train and obtain the safety model using the improved safety loss. In this process, a straightforward oversampling strategy is used to address data distribution imbalances. 3) Collaborative Reasoning. Employ the same collaborative decoding strategy as CoSec to improve the target model security. For the sake of completeness of content, we describe the three phases of CoSec+ in detail in this section.

A. Functional Correctness Alignment

In order to address the problem of discrepancy between the functional correctness of the safety model and the target model in CoSec, which reduces the functional correctness of the target model, we try to utilize knowledge distillation in CoSec+ in order to calibrate the functional correctness of the safety model (student model). Specifically, we performed knowledge distillation on the student model using the code model with the largest parameters in the same model family and the Magicoder dataset [49]. During the knowledge distillation process, we employed the rKLD loss function to enhance the student model's functional accuracy. This encourages the student to approximate the teacher's preferred distribution within its own capacity, rather than forcing it to fully replicate the teacher's output distribution. Additionally, we introduced a cross-entropy loss to enable the student model to learn directly from the correct answers, thereby preventing it from mimicking potential error patterns in the teacher model. We optimize the student model using a combined loss function weighted by rKLD loss and cross-entropy loss as follows:

$$\mathcal{L}_{rkl} = -\sum_{i=1}^{|n|} q_\theta(x_i|X_{1:i-1}) \log\left(\frac{q_\theta(x_i|X_{1:i-1})}{p(x_i|X_{1:i-1})}\right) \quad (3)$$

$$\mathcal{L}_{lm} = -\sum_{i=1}^{|n|} \log q_\theta(x_i|X_{1:i-1}) \quad (4)$$

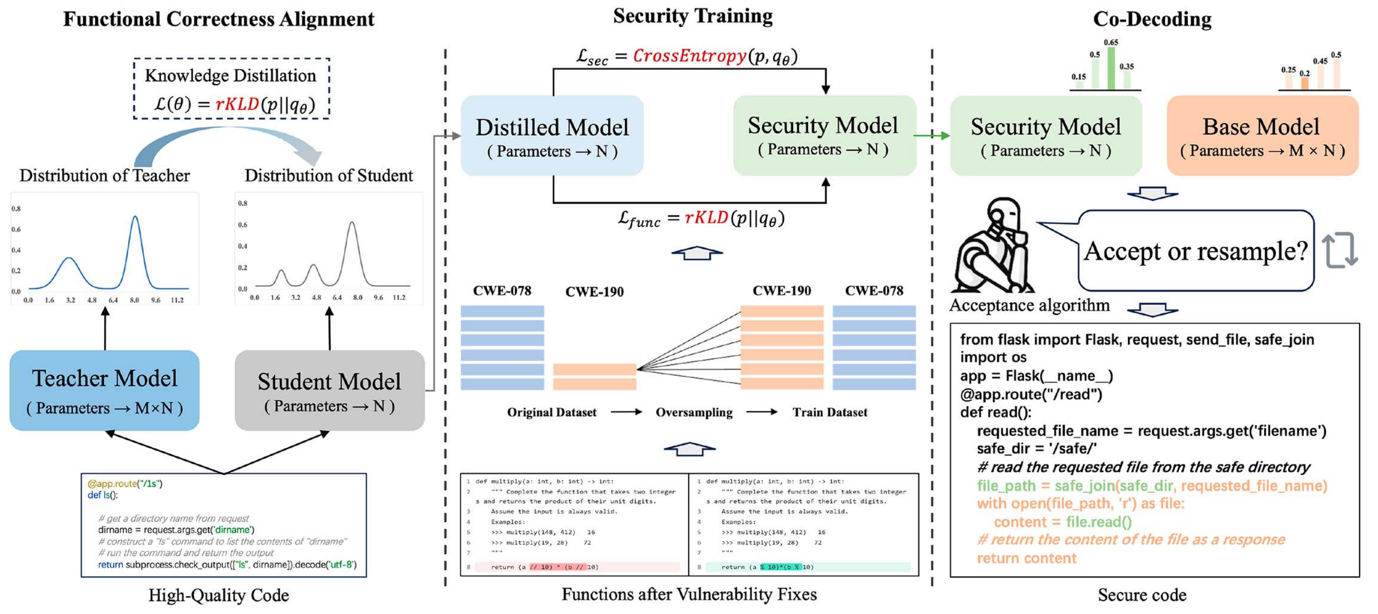
$$\mathcal{L}_{kd} = \lambda\mathcal{L}_{rkl} + (1-\lambda)\mathcal{L}_{lm} \quad (5)$$

where n is the length of the sample, q_θ is the predictive distribution at the current time step, p is the predictive distribution of the teacher's model, and λ is the ratio of $\mathcal{L}_{rkl}$.

B. Security Training

To further improve the effectiveness of security hardening, in CoSec+, we oversample the training data,

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.



```
@app.route("/1s")
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')
```

```
def multiply(a: int, b: int) -> int:
    """ Complete the function that takes two integer
s 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)
```

```
def multiply(a: int, b: int) -> int:
    """ Complete the function that takes two integer
s 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)
```

```
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 content of the file as a response
    return content
```

Fig. 3. The overall framework of CoSec+.

and to prevent different training losses from affecting the effectiveness, we optimize the loss function in security training stage of CoSec.

1) Parameter and Data Efficient: We use Low-Rank Adaptation (LoRA) [50] to reduce the training as well as deployment cost of security models. LoRA is trained by first freezing all the parameters of the security model and then performing a low-rank substitution of the internal weight matrix, which in this case can be a fully-connected layer, an attention layer, or a mapping layer, to name a few. Where the low-rank matrix can be represented by $\delta W = W_{down} W_{up}$ with $W_{\text{down}} \in \mathbb{R}^{d\times r}$, $W_{up} \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$. In this way, the training parameters are only 0.11% of the original model.

2) Data Balancing: To cope with the imbalance in the training data, we use a straightforward oversampling strategy. Specifically, for the training data of different types of CWE, we compute the average number n (where n is 60) and use it as a target to achieve balance. For types with less than the average number, we replicate the samples in them until they are equal to n. For different languages of the same CWE type, we randomly replicate the programming languages with a smaller share to reach k (we set k to 10).

3) Security Fine-Tuning: Following He and Vechev [23], we utilize the collected pairs of before and after vulnerability fixing functions as a training set for the security model. Specifically, they find that the modified portion of the fix determines whether the code is secure or not, while the unmodified portion is associated with functional correctness. Differently, we obtain the security model simply by having the model learn the security-relevant context (i.e., the code snippet that was changed in the security fix) of the data through a weighted multitask loss function. We utilize weighted cross-entropy loss to allow the model to learn safe code patterns:

$$\mathcal{L}_{\text{sec}} = -\sum_{i=1}^{|\mathbf{n}|} m_i \cdot \log q_\theta \left(x_i \mid \mathbf{X}_{1:i-1}\right) \tag{6}$$

where m_i represents the mask applied to the loss term, specifying whether the token in the current time step is classified as a secure change. A value of 1 indicates a secure change, while 0 indicates otherwise. This approach promotes the model for learning safer code.

Unlike CoSec, we use the rKLD consistent with the functional correctness alignment phase to limit excessive deviations between the representation learned from the security model and the original representation. Based on these findings, our weighted security loss is modeled as

$$\mathcal{L}_{\text{func}} = \sum_{i=1}^{|\mathbf{n}|} (1 - m_i) \mathcal{S}\left(x_i \mid \mathbf{X}_{1:i-1}\right) log \left(\frac{\mathcal{S}\left(x_i \mid \mathbf{X}_{1:i-1}\right)}{\mathcal{B}\left(x_i \mid \mathbf{X}_{1:i-1}\right)}\right) \tag{7}$$

Where S denotes the predictive distribution of the security model at the current time step, and B denotes the predictive distribution of the base model. Taking the $1 - m_i$ implies that the loss function is only valid for unchanged encodings.

Finally, our multitasking loss function is defined as follows:

$$\mathcal{L} = \mathcal{L}_{\text{sec}} + \mathcal{L}_{\text{func}} \tag{8}$$

C. Supervised Co-Decoding

Our co-decoding process stems from the insight that in intelligent coding tool-assisted programming, human programmers manually intervene in the writing of the next (next few) characters once they realize that the auto-completion does not match the intent, and LLMs give new auto-completions based on the new code above. Meanwhile, models fine-tuned on secure data have a higher confidence in their predictions. When the relative confidence of the safe model with respect to the base model is above a threshold, it is a safe prediction; otherwise, we use the target model for sampling to keep the code correct. The procedure of co-decoding is shown in Algorithm 1. Specifically,

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

when LLMs get a code snipet $x_1, \cdots, x_{n-1}, x_n$, the security model and the target model synchronize to generate token x_{n+1} at each time step until a stop sign is encountered or the maximum length is reached. We first take the most likely token $\tilde{x}$ predicted by the security model at each time step as a candidate. Then, we compute the probability $B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x})$ of this candidate token in the target model as well as the probability $S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x})$ in the security model. Finally, we use the acceptance algorithm to evaluate whether the security model's prediction should be accepted as the output of the current time step, i.e., if the ratio of the probability of $\tilde{x}$ in the target model to the probability in the security model is greater than the acceptance threshold a, then it is accepted. The acceptance algorithm is shown below:

$$a < \min\left(1, \frac{B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x})}{S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x})}\right) \tag{9}$$

Namely, for a token predicted by the security model, if its confidence level surpasses that of the base model by a specified threshold (similar to CoSec, we set this value to the default of 0.3), it is highly likely that this token meets the desired security requirements.

D. Supervised Co-Decoding

Algorithm 1: Co-Decoding

```
**Code prompt:** $x_1, \cdots, x_n$
**Max length:** $L$; **Number of samples:** $m$;
**Terminator:** EOS; **acceptance threshold:** $a$
**while** *true* **do**
    **if** $x_{n+1} == EOS \ or \ n \geq L$ **then**
        break
    **else**
        Obtain model output:
        $logits^s_{1:m} \leftarrow \text{Security}(x_1, \ldots, x_{n-1})$
        $logits^b_{1:m} \leftarrow \text{Base}(x_1, \ldots, x_{n-1})$
        Obtain m probabilities:
        $S(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_{1:m})$
        $B(x_{n+1} \mid x_1, \ldots, x_n, \tilde{x}_{1:m})$
        Obtain m security tokens:
        $\tilde{x}_{1:m} \leftarrow \text{multinomial}(\text{softmax}(logits^s_{1:m}))$
        **for** $i = 1$ ***to*** $m$ **do**
            **if** $a < \min\left(1, \frac{B(x_{n+1}|x_1,\ldots,x_n,\tilde{x}_i)}{S(x_{n+1}|x_1,\ldots,x_n,\tilde{x}_i)}\right)$ **then**
                Accept security model's prediction:
                $x_{n+1} \leftarrow \tilde{x}_i$
            **else**
                Resample from the base model:
                $x_{n+1} \leftarrow \text{multinomial}(\text{softmax}(logits^b_i))$
            **end**
            $[x_1, \ldots, x_n]_i \leftarrow [x_{n+1}]_i$
        **end**
    **end**
**end**
```

V. Experimental Settings

In this section, we introduce the target models, baselines and dataset, followed by a summary of the evaluation metrics used in the experiment. We then provide a detailed overview of the implementation specifics. The experiments are designed to address the following research questions:

- **RQ3:** How well does CoSec+ perform in security hardening?
- **RQ4:** How well does CoSec+ perform in maintaining functional correctness?
- **RQ5:** Is CoSec+ also effective for vulnerability types not seen in the training set?
- **RQ6:** How different components affect security and functional correctness?

A. Target Models

- **CodeGen** [4]. CodeGen-multi is a standard autoregressive language model for multi-turn program synthesis. It has been extensively trained on large-scale datasets, allowing it to be coded across different scenarios. The dataset is oriented towards C, C++, Go, Java, JavaScript, and Python [51]. CodeGen-multi is available in four different model sizes: 350M, 2.7B, 6.1B, and 16.1B.
- **StarCoderBase** [36]. StarCoderBase is a language model designed specifically for code generation tasks. It is trained with causal language modeling as well as fill-in-the-middle (FIM) objectives [52] on TheStack (v1.2) [53]. StarCoderBase can generate code snippets that integrate seamlessly into existing codebases. The model comes in four sizes with varying parameter counts to suit different application needs: 1B, 3B, 7B, and 15.5B.
- **DeepSeek-Coder** [6]. DeepSeek-Coder is developed by DeepSeek AI. The training data consists of 87% code and 13% natural language text. Its training data consists of 87% code and 13% natural language text, of which the natural language is mainly English and Chinese content. DeepSeek-Coder is offered in multiple sizes to accommodate different performance requirements and computational resources: 1.3B, 5.7B, 6.7B, and 33B.
- **Qwen2.5-Coder** [54]. Qwen2.5-Coder is released by Alibaba. It is trained on 5.5 trillion tokens and supports a context length of up to 128K. The series includes multiple model sizes: 0.5B, 1.5B, 3B, 7B, 14B, and 32B.

B. Datasets

- We use the evol-codealpaca-v1 dataset as the training dataset of knowledge distillation, which was proposed by Wei et al. [49] to generate more diverse, realistic, and controllable code using open-source code snippets to inspire LLM. The dataset was formed using the Evol-Instruct method by augmenting code snippets from 80,000 documents filtered in StarCoderData and 75,000 data synthesized by GPT-4-0613. We used 10k entries as the test set, with the remaining data serving as the training set.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

TABLE III
DIFFERENCES IN VULNERABILITY TYPES BETWEEN THE OLD AND NEW UNSEEN TEST SETS

Type	Explain	Old Set	New Set
CWE-119	Buffer Errors	✓	✓
CWE-502	Insecure Deserialization	✓	✓
CWE-732	Insecure Permissions	✓	✓
CWE-798	Use of Hard-coded Credentials	✓	✓
CWE-020	Input Validation Error	✗	✓
CWE-117	Log Injection	✗	✓
CWE-777	Regex without Anchors	✗	✓

- We adopt the same secure training dataset as in Section III to train the security model of CoSec+, which is proposed by He and Vechev [23].

C. Evaluation Frameworks

We use the same evaluation framework as in section III to test the security and functional correctness provided by CoSec+ for the target model. These two evaluation frameworks are proposed by Chen et al. [48] and Pearce et al. [20] respectively. Additionally, we obtained three new, previously unseen vulnerability types from [55] to more comprehensively evaluate the effectiveness of CoSec+. The newly added types are shown in Table III. The newly added vulnerability types include CWE-020 (Input Validation Error), CWE-117 (Log Injection), and CWE-777 (Regex without Anchors). According to the MITRE report, all of them are classified as high-risk vulnerabilities and should be prioritized for fixing.

D. Implementation Details

We compare CoSec+ with the original models, LoRA fine-tuning [50], and CoSec [24]. All code LLMs and their corresponding tokenizers used in our experiments were loaded from the official Huggingface repository [56], with hyperparameter settings detailed in Table IV.

In the Functional Correctness Alignment phase, for four different model families, we select the model with smaller parameters as the student model, i.e., CodeGen-350M, StarCoderBase-1B, DeepSeek-Coder-1.3B, and Qwen2.5-Coder-1.5B and the model with larger parameters as the teacher model, i.e., CodeGen-6.1B, StarCoderBase-7B, DeepSeek-Coder-6.7B, and Qwen2.5-Coder-14B. We set the hyperparameters to a maximum of 3 epochs, a batch size of 24, a learning rate of 1e-5, a Clip Gradient of 1.0, a λ of 0.5, a Temperature of 1.0 and a maximum input text length of 1024.

We performed security training on four distilled models (i.e., CodeGen-350M, StarCoderBase-1B, DeepSeek-Coder-1.3B, and Qwen2.5-Coder-1.5B) from the first step (i.e., Functional Correctness Alignment) as a base for the security model. Our training hyperparameters for this phase are consistent with CoSec [24], i.e., setting the LoRA with a rank of 8, an alpha of 16, and a dropout rate of 0.1 to achieve parameter efficiency.

TABLE IV
HYPERPARAMETER SETTINGS

Hyperparameter	Value	Hyperparameter	Value
Knowledge Distillation Settings			
Optimizer	AdamW	Learning Rate	1e-5
Traing Batch Size	24	Max.Tokens	1024
Gradient Acc Step	1	Training Epoch	3
Clip Gradient	1.0	λ	0.5
Temperature	1.0		
Security 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	16
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
Sample Temp	0.4	Accept Threshold	0.3

The maximum input length was maintained at 1024, the maximum number of epochs was limited to 8, the learning rate was set to 5e-5. We take the round with the smallest loss value during training as the best checkpoint.

During the inference phase for RQ3, RQ4, RQ5, and RQ6, we fixed the number of sampling iterations to 25, the maximum number of newly generated tokens to 256, TOP-P to 0.95, the minimum prediction probability to 0, the acceptance threshold to 0.3, and the sampling temperatures for both the security model and the target base model to 0.4, as detailed in Table IV.

All experiments were done using the PyTorch framework on an Ubuntu server with 2 NVIDIA A800 GPU.

VI. EXPERIMENTAL RESULTS

A. RQ3:How Well Does CoSec+ Perform in Security Hardening?

1) Methodology: To comprehensively evaluate the security of CoSec+, we adopt the benchmark proposed by Pearce et al. [20] to assess the security of our method, and report both the overall security rate and the relative percentage improvement in security. We apply CoSec+ to enhance the security of target base models with eight different parameter sizes across four model families, and compare it with LoRA and CoSec.

2) Result: The experimental results are shown in Table V. For the CodeGen-350M model, CoSec+ achieved a 28.3% relative security improvement, which is 6.1% higher than CoSec's 22.2%. This increase highlights the effectiveness of CoSec+ in enhancing the security of smaller parameter models. For the larger CodeGen-2.7B model, CoSec+ achieved a significant improvement of 37.7%, greatly surpassing CoSec 19.7% increase. For the CodeGen-6.1B model, which already has a relatively high baseline security rate of 68.2%, CoSec+ achieved

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

TABLE V
IMPACT OF COSEC+ ON SECURITY. DARKER SHADED INDICATE THE BEST SECURITY PERFORMANCE UNDER THE SAME CWE CATEGORY, WITH LIGHTER SHADES REPRESENTING THE SECOND

Model	Type	Security Radio (CWE)									Total
		- 089 -	- 125 -	- 078 -	- 476 -	- 416 -	- 022 -	- 787 -	- 079 -	- 190 -	
CodeGen -350M	Origin	85.4%	49.5%	37.2%	7.6%	81.8%	29.8%	62.9%	48.2%	96.0%	55.4%
	+LoRA	100.0%	60.2%	89.7%	63.7%	63.7%	100.0%	87.7%	100.0%	100.0%	85.0%
	+CoSec	87.6%	48.0%	58.7%	53.3%	65.7%	50.0%	84.8%	66.7%	89.6%	67.1%
	+CoSec+	96.0%	44.0%	85.7%	38.1%	80.5%	54.2%	76.7%	69.0%	95.9%	71.1%
		↑12.4%	↓11.1%	**↑130.4%**	**↑401.3%**	↓1.6%	**↑81.9%**	**↑21.9%**	**↑43.2%**	↓0.1%	**↑28.3%**
CodeGen -2.7B	Origin	84.0%	81.0%	11.2%	6.3%	100.0%	63.5%	52.0%	24.6%	81.3%	56.0 %
	+LoRA	100.0%	95.5%	44.9%	36.8%	100.0%	87.4%	63.1%	72.2%	82.0%	75.7%
	+CoSec	66.5%	72.9%	52.8%	32.2%	82.6%	87.7%	56.5%	65.4%	78.0%	66.1%
	+CoSec+	91.7%	59.1%	81.0%	67.5%	83.4%	85.9%	65.9%	70.0%	90.0%	77.1%
		↑9.2%	↓27.0%	**↑623.2%**	**↑971.4%**	↓16.6%	**↑35.3%**	**↑26.7%**	**↑184.6%**	**↑10.7%**	**↑37.7%**
CodeGen -6.1B	Origin	95.9%	80.9%	30.9%	20.5%	87.5%	90.1%	49.8%	62.3%	96.0%	68.2%
	+LoRA	100.0%	97.1%	57.0%	6.5%	97.1%	100.0%	79.2%	95.0%	98.0%	81.1%
	+CoSec	75.0%	64.7%	51.6%	29.1%	83.3%	100.0%	65.1%	96.9%	90.9%	72.9%
	+CoSec+	79.2%	63.5%	74.5%	32.7%	89.5%	100.0%	74.2%	92.1%	83.4%	76.6%
		↓17.4%	↓21.5%	**↑141.1%**	**↑59.5%**	**↑2.3%**	**↑11.0%**	**↑49.0%**	**↑47.8%**	↓13.1%	**↑12.3%**
StarCoder Base-1B	Origin	76.0%	64.6%	42.1%	12.4%	92.5%	51.5%	96.0%	60.4%	72.2%	63.7%
	+LoRA	100.0%	66.7%	63.0%	30.4%	92.0%	55.8%	73.9%	93.4%	53.9%	69.9%
	+CoSec	90.0%	83.4%	34.5%	35.4%	76.1%	80.0%	97.6%	88.1%	75.0%	73.3%
	+CoSec+	100.0%	85.7%	69.1%	17.4%	100.0%	61.8%	100.0%	86.1%	84.0%	78.2%
		↑31.6%	**↑32.7%**	**↑64.1%**	**↑40.3%**	**↑8.1%**	**↑20.0%**	**↑4.2%**	**↑42.5%**	**↑16.3%**	**↑22.8%**
StarCoder Base-7B	Origin	88.0%	90.9%	29.6%	20.9%	97.9%	95.0%	46.5%	100.0%	72.9%	71.3%
	+LoRA	100.0%	85.0%	81.6%	29.6%	100.0%	94.1%	30.2%	100.0%	91.7%	79.1%
	+CoSec	89.6%	72.8%	54.0%	23.8%	100.0%	51.5%	97.6%	100.0%	68.0%	73.0%
	+CoSec+	100.0%	83.4%	57.1%	26.3%	100.0%	100.0%	100.0%	100.0%	92.0%	84.3%
		↑13.6%	↓8.3%	**↑92.9%**	**↑25.8%**	**↑2.1%**	**↑5.3%**	**↑115.1%**	**-**	**↑26.1%**	**↑18.2%**
DeepSeek-Coder-1.3B	Origin	92.0%	75.0%	25.0%	10.4%	100.0%	82.5%	66.0%	100.0%	68.8%	68.9%
	+LoRA	100.0%	100.0%	100.0%	50.5%	100.0%	92.3%	64.0%	100.0%	80.0%	87.4%
	+CoSec	100.0%	83.4%	40.7%	52.0%	100.0%	91.7%	68.8%	100.0%	80.0%	79.6%
	+CoSec+	100.0%	100.0%	31.4%	29.4%	100.0%	100.0%	93.9%	100.0%	70.0%	80.5%
		↑8.7%	**↑33.3%**	**↑25.6%**	**↑182.7%**	**-**	**↑21.2%**	**↑42.3%**	**-**	**↑1.7%**	**↑16.8%**
DeepSeek-Coder-6.7B	Origin	82.0%	79.2%	55.8%	2.0%	100.0%	97.2%	42.0%	90.0%	97.6%	71.8%
	+LoRA	96.0%	88.0%	100.0%	0.0%	100.0%	100.0%	60.6%	100.0%	97.9%	82.5%
	+CoSec	96.0%	77.1%	36.0%	2.3%	100.0%	95.0%	74.0%	100.0%	94.0%	74.9%
	+CoSec+	98.0%	90.0%	57.6%	12.9%	100.0%	100.0%	91.7%	100.0%	88.0%	82.0%
		↑19.5%	**↑13.6%**	**↑3.2%**	**↑545.0%**	**-**	**↑2.9%**	**↑118.3%**	**↑11.1%**	↓9.8%	**↑14.2%**
Qwen2.5-Coder-14B	Origin	64.6%	90.9%	80.2%	2.4%	100.0%	83.4%	62.5%	100.0%	100.0%	76.0%
	+LoRA	100.0%	100.0%	88.0%	0.0%	100.0%	100.0%	91.7%	100.0%	100.0%	86.6%
	+CoSec	68.0%	80.4%	36.7%	6.6%	100.0%	100.0%	54.0%	100.0%	84.0%	70.0%
	+CoSec+	80.0%	83.4%	47.6%	8.0%	100.0%	97.9%	75.0%	100.0%	97.9%	76.6%
		↑23.8%	↓8.3%	↓40.6%	**↑233.3%**	**-**	**↑17.4%**	**↑20.0%**	**-**	↓2.1%	**↑0.8%**

an 12.3% relative security improvement. This demonstrates that CoSec+ can effectively enhance security even in models with a robust security foundation. Regarding the suboptimal performance of CoSec+ on CWE-125 and CWE-416, we attribute this to the limited knowledge capacity of the 350M security model, which was insufficient to effectively learn secure coding patterns. As we all know, security fine-tuning merely adjusts the way a model expresses the knowledge it has acquired during pretraining [57]. Experimental results on StarCoderBase and DeepSeek-Coder confirm this observation.

We also evaluate the hardening capability of CoSec+ on the StarCoderBase series and DeepSeek-Coder series. For StarCoderBase-1B, CoSec+ achieved a 22.8% relative security improvement, 7.7% higher than CoSec. For StarCoderBase-7B, CoSec+ improves security by 18.2%, significantly surpassing CoSec's 2.4% with a 15.8% increase. Within the DeepSeek-Coder series, CoSec+ delivers a 16.8% relative security improvement for the 1.3B model, surpassing CoSec. On the 6.7B model, CoSec+ achieves a notable 14.2% gain—4.5% higher than CoSec—demonstrating its ability to address complex security vulnerabilities that CoSec may have missed.

To further evaluate the effectiveness of the CoSec+ pipeline on more advanced and larger-size models, we use the 1.5B Qwen2.5-Coder as the security model to provide security hardening for the Qwen2.5-Coder-14B. Experimental results show

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

that CoSec+ is still able to deliver a 0.8% improvement in security for the larger model.

Compared to directly applying LoRA fine-tuning on the target model using security-related data, our method achieves comparable performance without accessing the model's internal parameters—and even outperforms it on certain models. For example: CoSec+ achieves relative improvements of 1.8%, 11.9%, 6.6%, and 12.6% on CodeGen-2.7B, StarCoderBase-1B, StarCoderBase-7B, and DeepSeek-Coder-6.7B, respectively.

It is worth emphasizing that the pretraining settings differ across models of different scales, even within the same family. As a result, the models exhibit different confidence levels for the same token. Therefore, the final security improvement achieved through co-decoding cannot reach the same level across all models.

Answer to RQ3: CoSec+ can effectively guide code LLMs with different parameters to generate more secure code in real-world usage scenarios. Specifically, our framework improves the average relative security rate of seven code LLMs by 0.8% to 37.7%. Compared to CoSec, it achieves an additional gain of 1.3% to 19.7%. Furthermore, CoSec+ outperforms LoRA by 1.8% to 12.6% on certain models.

B. RQ4:How Well Does CoSec+ Perform in Maintaining Functional Correctness?

1) Methodology: We naively use HumanEval [48] to measure the impact of CoSec+ on the functional correctness of the target base models after security hardening. The experiments involve eight models of varying sizes across five model families, including CodeGen, StarCoderBase, DeepSeek-Coder, and Qwen2.5-Coder. We focus on the case of Pass@1, Pass@5, and Pass@10 for the consideration of response time and developer's efficiency in real application scenarios.

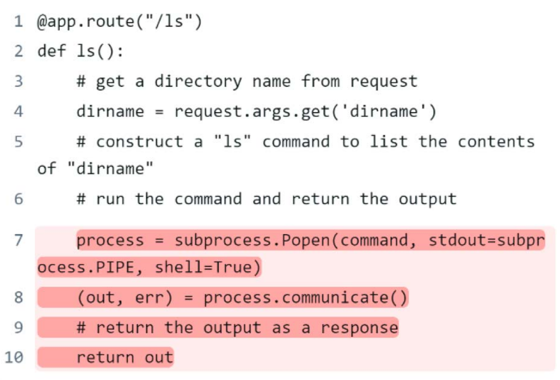
2) Result: Table VI provides a comprehensive summary of the original functional correctness of the target base models, as well as the functional correctness after applying LoRA, CoSec, and CoSec+. It can be seen that for the CodeGen series, CoSec+ provides a 23.4% improvement in Pass@1 for the 350M model. Compared to LoRA fine-tuning, we achieve a 43.6% improvement. However, for the 2.7B and 6.1B models, it causes some decline. Nevertheless, through functional correctness alignment, CoSec+ still achieves better functional correctness preservation for CodeGen-2.7B and -6.1B. We attribute this phenomenon to the fact that the 350M model has too few parameters, and distillation does not bring a significant leap in code capabilities. This viewpoint is supported by experiments on StarCoderBase and DeepSeek-Coder. For the StarCoderBase series, CoSec+ provides Pass@1 improvements of 51.1% and 21.0% for the 1B and 7B models, respectively. Compared to LoRA fine-tuning, CoSec+ offers improvements of 55.8% and 8.0% in Pass@1, significantly outperforming CoSec. Similarly, CoSec+ also provides functional correctness improvements for the DeepSeek-Coder series. Specifically, CoSec+ provides a 21.1% improvement in Pass@1 for the 1.3B model and a 0.7% improvement for the 6.7B model. Compared to LoRA fine-tuning, CoSec+ achieves relative Pass@1 improvements of 18.6% and 3.8%, respectively—both significantly higher than those provided by CoSec in terms of functional correctness. For Pass@5 and Pass@10, CoSec+ provides relative improvements ranging from 20.6% to 76.4% for the StarCoderBase series and from 5.2% to 26.2% for the DeepSeek-Coder series. Furthermore, for advanced models with more than 7B parameters, such as Qwen2.5-Coder-14B, CoSec+ achieves an improvement of approximately 0.1% on Pass@5 and Pass@10, with a decrease on Pass@1 of no more than 1%.

TABLE VI
IMPACT OF COSEC+ ON FUNCTIONAL CORRECTNESS

Model	**Type**	**HumanEval**		
		Pass@1	Pass@5	Pass@10
CodeGen -350M	Origin	6.4	9.5	10.9
	+LoRA	5.5	8.5	10.0
	+CoSec	5.5	8.4	9.7
	CoSec+	**7.9**	**11.9**	**13.3**
CodeGen -2.7B	Origin	12.9	20.8	23.6
	+LoRA	11.6	20.0	23.9
	+CoSec	10.1	16.4	20.2
	+CoSec+	12.8	20.4	23.2
CodeGen -6.1B	Origin	17.7	24.6	27.4
	+LoRA	17.3	24.6	27.5
	+CoSec	13.4	20.9	25.4
	+CoSec+	14.1	21.9	25.8
StarCoder Base-1B	Origin	13.3	19.9	22.9
	+LoRA	12.9	21.4	26.1
	+CoSec	13.1	19.4	22.1
	+CoSec+	**20.1**	**34.2**	**40.4**
StarCoder Base-7B	Origin	20.0	31.6	36.0
	+LoRA	22.4	34.5	38.5
	+CoSec	17.3	27.8	33.1
	+CoSec+	**24.2**	**38.1**	**43.8**
DeepSeek -Coder-1.3B	Origin	27.9	42.0	46.9
	+LoRA	28.5	44.4	50.3
	+CoSec	27.8	44.6	51.1
	+CoSec+	**33.8**	**52.0**	**59.2**
DeepSeek -Coder-6.7B	Origin	43.7	63.0	69.2
	+LoRA	42.4	63.2	69.5
	+CoSec	38.5	60.3	67.1
	+CoSec+	**44.0**	**66.3**	**73.4**
Qwen2.5 -Coder-14B	Origin	71.1	85.1	88.0
	+LoRA	72.9	84.7	87.6
	+CoSec	67.3	84.3	88.2
	+CoSec+	70.8	**85.3**	**88.8**

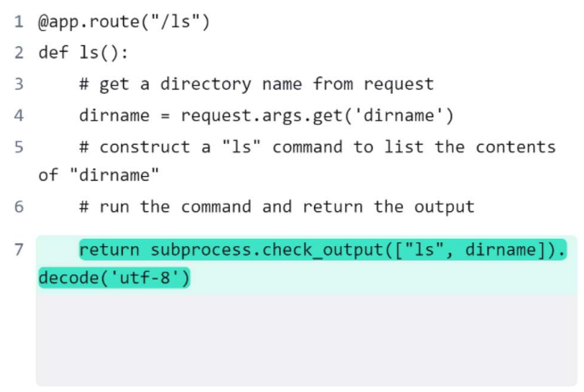
We attribute this improvement to two reasons: 1) rKLD focuses more on the high probability region of the teacher's predictive distribution than KLD, which is beneficial for tasks like code generation that seeks for accuracy (We will further discuss the advantages of rKLD over KLD in Section VII.); 2) the standard cross entropy motivates the student model to focus on the correct answer in the same way. Fig. 4(a) and 4(b) show examples of code generated by the model after security hardening through CoSec+. It can be seen that the code generated before security hardening has a CWE-078 vulnerability, while the code does not run properly. The code after security hardening not only realizes the security hardening, but also can run correctly.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.



```
@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=subprocess.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

Fig. 4. An illustration of functional correctness improvement.

Answer to RQ4: CoSec+ is able to maintain or even surpass the functional correctness of the target models. Specifically, CoSec+ provides a relative improvement of 0.7% to 51.1% on Pass@1 while performing security hardening on the CodeGen, StarCoderBase,and DeepSeek-Coder. Relative improvements of 0.4% to 49.3% were also observed in Pass@5 and Pass@10. In contrast, the slight performance drops observed on Qwen2.5-Coder-14B are considered acceptable.

C. RQ5:Is CoSec+ Also Effective for Vulnerability Types Not Seen in the Training Set?

1) Methodology: To answer this question, we evaluate the effectiveness of CoSec+ for security hardening on 7 CWEs that did not appear in the security training dataset [20]. Considering the model capacity and security hardening difficulty, we report the security hardening effectiveness of the larger models of each of the three model families.

2) Result: From the Table VII, it can be seen that although the security model has not encountered these vulnerability types before, CoSec+ still demonstrates strong generalization capability. Specifically, for CodeGen-2.7B, applying our hardening method leads to significant improvements in five vulnerability types: CWE-119, CWE-502, CWE-798, CWE-020, and CWE-117. Among them, for CWE-117 and CWE-502, improvements of 100% and 139.4% were realized, respectively. The overall security of CodeGen-2.7B increased from 40.5% to 52.6%, representing a relative improvement of 29.6%. In the case of StarCoderBase-7B, CoSec+ achieves security improvements for five vulnerability types: CWE-119, CWE-502, CWE-020, CWE-117, and CWE-777. Among them, the improvement for three types, including CWE-502, CWE-117, and CWE-777, is significant, reaching 97.8%-100%. The security of StarCoderBase-7B improved from 46.0% to 49.0%, achieving a relative increase of 6.4%. For DeepSeek-Coder-6.7B, the application of our hardening method results in substantial improvements across all elevated CWE categories. CWE-119 performance increases from 42.2% to 53.3%, yielding a 26.3% enhancement. CWE-502 reached a perfect score of 100.0%, up from 91.4%, an improvement of 9.4%. CWE-732 increased from 86.1% to 97.3%, a significant improvement of 13.0%. CWE-020 improved from 52.5% to 58.4%, a relative increase of 11.1%. CWE-777 increased from 0.0% to 6.3%. As a result, the overall code security score improved from 48.0% to 54.2%, a remarkable 12.9% increase.

Answer to RQ5: CoSec+ is capable of hardening the vulnerability types that has not shown in the training set of security models. Specifically, for previously unseen vulnerability types, it improves the security of CodeGen-2.7B by 29.7%, StarCoderBase-6B by 6.4%, and DeepSeek-Coder-6.7B by 12.9%. These results highlight the general applicability of our method in security hardening.

D. RQ6:How Different Components Affect Security and Functional Correctness?

1) Methodology: In order to explore the enhancement of CoSec+ by each of our proposed components, we performed ablation experiments on CoSec+ using the CodeGen series as backbone models. The ablation on three different parametric quantities of models is also intended to reflect the generalization capabilities of CoSec+. We sequentially removed knowledge distillation in functional correctness alignment, oversampling in security training, and rKLD loss in security training. In this, while removing each component, we keep the others for the purpose of controlling the variables. Finally, we report on safety as well as functional correctness after removal. Where the safety assessment is consistent with RQ3, reporting safety on the 9 categories of CWE seen in the training set. Functional correctness is consistent with RQ4, reporting Pass@1, Pass@5, and Pass@10.

2) Result: As can be seen in Table VIII, knowledge distillation is critical for functional correctness after deploying CoSec+. When knowledge distillation was removed, the Pass@1 scores for CodeGen-350M, CodeGen-2.7B, and CodeGen-6.1B decreased by 41.0%, 26.6%, and 15.6%, respectively. Similarly, for Pass@5 and Pass@10, we observed reductions ranging from 8.1% to 50.0%.

To assess the importance of oversampling for CoSec+, we proceeded to remove this operation. It can be seen that

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

TABLE VII
SECURITY HARDENING EFFECTS OF COSEC+ ON CWES NOT PRESENT IN TRAINING DATA

Model	Size	CWE-119	CWE-502	CWE-732	CWE-798	CWE-020	CWE-117	CWE-777	Total
CodeGen	2.7B	56.2%	39.3%	69.8%	37.4%	73.9%	0.0%	7.2%	40.5%
+Harden	350M	65.6%	94.1%	44.9%	48.4%	97.8%	13.0%	4.4%	52.6%
		↑16.7%	**↑139.4%**	↓35.7%	**↑29.4%**	**↑32.3%**	**↑100.0%**	↓39.2%	**↑29.7%**
StarCoderBase	7B	59.8%	49.8%	86.7%	75.9%	50.0%	0.0%	0.0%	46.0%
+Harden	1B	67.8%	98.5%	59.7%	49.7%	54.6%	8.7%	4.0%	49.0%
		↑13.4%	**↑97.8%**	↓31.1%	↓34.5%	**↑9.1%**	**↑100.0%**	**↑100.0%**	**↑6.4%**
DeepSeek-Coder	6.7B	42.2%	91.4%	86.1%	63.7%	52.5%	0.0%	0.0%	48.0%
+Harden	1.3B	53.3%	100.0%	97.3%	64.1%	58.4%	0.0%	6.3%	54.2%
		↑26.3%	**↑9.4%**	**↑13.0%**	**↑0.6%**	**↑11.1%**	**-**	**↑100.0%**	**↑12.9%**

TABLE VIII
EFFECTIVENESS OF EACH COMPONENTS

Security Model	**Target Base LLMs**	**Components**	**Code Security**	**HumanEval**		
				Pass@1	Pass@5	Pass@10
CodeGen-350M	CodeGen-350M	Cosec+	71.1%	10.0	16.6	19.8
		without KD	70.8%(↓0.4%)	5.9(↓41.0%)	8.9(↓46.4%)	9.9(↓50.0%)
		without Oversampling	70.4%(↓1.0%)	8.6(↓14.0%)	12.2(↓26.5%)	13.5(↓31.8%)
		replace rKLD as KLD	68.4%(↓3.8%)	8.2(↓18.0%)	12.0(↓27.7%)	13.4(↓32.3%)
	CodeGen-2.7B	Cosec+	77.1%	12.8	20.4	23.2
		without KD	70.9%(↓8.0%)	9.4(↓26.6%)	15.6(↓23.5%)	18.8(↓19.0%)
		without Oversampling	68.7%(↓10.9%)	11.8(↓7.8%)	19.1(↓6.4%)	21.9(↓5.6%)
		replace rKLD as KLD	77.9%(↑1.0%)	12.6(↓1.6%)	20.2(↓1.0%)	23.1(↓0.4%)
	CodeGen-6.1B	Cosec+	76.6%	14.1	21.9	25.8
		without KD	76.7%(↑0.1%)	11.9(↓15.6%)	19.7(↓10.0%)	23.7(↓8.1%)
		without Oversampling	74.6%(↓2.6%)	13.4(↓5.0%)	20.6(↓5.9%)	24.0(↓7.0%)
		replace rKLD as KLD	75.8%(↓1.0%)	13.8(↓2.1%)	21.4(↓2.3%)	24.9(↓3.5%)

CodeGen-350M, CodeGen-2.7B, and CodeGen-6.1B reduce security by 1.0%, 10.9%, and 2.6%, respectively. The effect of oversampling on functional correctness we attribute to the fact that this operation not only allows the model to learn the laws of security for sparse CWE types multiple times, but also the laws of functional correctness in these cases.

Finally, we analyze whether $\mathcal{L}_{func}$ in security training needs to be consistent with that in knowledge distillation. It can be observed that when the rKLD loss function is replaced with the KLD loss function, the Pass@1 scores for CodeGen-350M, CodeGen-2.7B, and CodeGen-6.1B decreased by 18.0%, 1.6%, and 2.1%, respectively. Similarly, for Pass@5 and Pass@10, we observed reductions ranging from 0.4% to 32.3%.

Answer to RQ6: In summary, knowledge distillation, oversampling, and replacing KLD with rKLD help security models learn functionally correct predictions consistent with larger parameter models in the same family as well as higher security. Specifically, removing knowledge distillation leads to a decrease in functional correctness after security hardening by 8.1% to 50.0%. The absence of oversampling and rKLD in the security training process leads to a 1.0% to 10.9% decrease in security and a 0.4% to 32.3% decrease in functional correctness.

VII. DISCUSSION

This section discusses the influence of rKLD in functional correctness alignment, the effects of CoSec+ on inference speed and the threats to validity.

A. rKLD or KLD

We first discuss the advantages of rKLD in aligning functional correctness. According to Gu et al. [25]: due to the more complex probability space of LLMs, rKLD places greater focus on the regions where the teacher model is most confident, resulting in better distillation performance compared to traditional KLD. We conducted comparative experiments using both rKLD and KLD with the same hyperparameters and teacher model settings. As shown in Table IX, rKLD achieves more improvements in code capabilities. Specifically, in the improvement of Pass@1 for Codegen-350M, Deepseek-Coder-1.3b, and StarCoderBase-1B, rKLD outperforms KLD by 5.4%, 9.3%, and 14.0%, respectively.

B. Effects on Inference Speed

Then, we discuss the impact of CoSec+ on inference speed. Due to differences in data types and Transformer architectures among different model families, the inference time also varies. We use the default settings provided on the official websites

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

TABLE IX
THE IMPACT OF FKLD AND RKLD ON THE FUNCTIONAL CORRECTNESS OF SECURITY MODELS

Model	Algorithm	Pass@1
CodeGen-350M	Origin	6.4
	KLD	13.0
	rKLD	**13.7**
		↑5.4%
DeepSeek-Coder-1.3b	Origin	27.9
	KLD	39.8
	rKLD	**43.5**
		↑9.3%
StarCoderBase-1B	Origin	13.3
	KLD	19.3
	rKLD	**22.0**
		↑14.0%

TABLE X
INFERENCE TIME COST PER 20 TOKENS

Method	Size	Time Cost (second)
CodeGen	350M	0.49
+ CoSec+	350M	0.50
CodeGen	6.1B	1.06
+ CoSec+	350M	1.25
DeepSeek-Coder	6.7B	0.64
+ CoSec+	1.3B	0.81
StarCoderBase	7B	0.51
+ CoSec+	1B	0.57
Qwen2.5-Coder	14B	1.15
+ CoSec+	1.5B	0.95

to sample one response for the same prompt and record the average time consumption per 20 tokens. As shown in Table X, after applying CoSec+, the model inference time consumption per 20 tokens increases by varying degrees, ranging from 0.01s to 0.2s. However, the overall response time remains within the acceptable range for users.[5] The widespread acceptance of slow-thinking models like DeepSeek-r1 also demonstrates that, when applying AI tools, generation quality is more important than fast response [58]. To provide secure code completion services as efficiently as possible, we have made the following efforts: On one hand, we use KV-Cache [59] to improve inference speed; on the other hand, our inference framework supports batch processing.

C. Threats to Validity

One threat to validity relates that our experiments focus on only two languages, Python and C/CPP. Although Python and C/CPP do not represent all programming languages, these two languages are two of the top three most popular programming languages. Moreover, the training and testing data in the experiments are carefully hand-crafted and of high quality, enabling reliable conclusions to be drawn. Another important threat comes from possible data leakage. Given that code LLMs often use public repositories such as GitHub for their pre-training data [4], [36], the pre-training data for the models we use may already be contained in the datasets we use. To answer this question, we analyze the Benchmark used for the tests studied in this paper. our investigations show that in the security tests, the prompts contain only manually defined prompts for different scenarios and do not contain a strictly defined ground-truth. After being generated, the security is analyzed and counted by a static scanning tool. HumanEval, on the other hand, is an authoritative testing framework introduced by OpenAI, and also does not contain a strict ground-truth. We are confident that our conclusions are not affected by any potential data leakage.

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

VIII. CONCLUSION AND FUTURE WORKS

In this paper, we reiterate the effectiveness of CoSec in security hardening of black-boxed code LLMs, while also pointing out areas for improvement in its security hardening and functional correctness maintenance. To address this problem, we propose a novel approach, CoSec+, through knowledge distillation and optimized security training. CoSec+ consists of three phases. In the first phase, the base of the security model is functionally correctly aligned to the model with the largest parameter in the family of that model using knowledge distillation. In the second phase, the security model is obtained using the improved security training. In this, an over-sampling is used to learn security laws on sparse CWEs and programming languages, and a replacement rKLD is used to mitigate interference from different loss functions. In the third stage, a very small independent security model completed by training is utilized to infer the next lexical element in an iterative manner with the target base model to achieve security reinforcement. Our inference framework does not require access to the internal parameters of the target model, providing a deployment scheme that takes into account both computational resources and the need for high-quality secure data. We conducted extensive experimental validation of our 8 models on 16 high-risk CWE vulnerabilities, and the results show that our approach is able to achieve strong security enhancements while maintaining or even improving target functional correctness.

Currently, due to tokenizer limitations, our approach only supports sharing between models of different sizes within the same model family. However, we are aware that not all large open-source models are released as families. Therefore, it is of interest to investigate a generalized security-enhancing approach that is applicable to different model architectures at decoding time in the future. In addition, comprehensively exploring the security of existing coding language models (both open- and closed-source) and constructing quality training data that covers a wider range of vulnerability types are also important research questions that we plan to address in the future.

REFERENCES

[1] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

[2] H. Touvron et al., “Llama 2: Open foundation and fine-tuned chat models,” 2023, *arXiv:2307.09288*.

[3] Z. Du et al., “GLM: General language model pretraining with autoregressive blank infilling,” in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, 2022, pp. 320–335.

[4] E. Nijkamp et al., “Codegen: An open large language model for code with multi-turn program synthesis,” 2022, *arXiv:2203.13474*.

[5] R. Li et al., “StarCoder: May the source be with you!” 2023, *arXiv:2305.06161*.

[6] DeepSeek. 2023. Deepseek coder: Let the code write itself. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-Coder

[7] B. Roziere et al., “Code Llama: Open Foundation Models for Code,” 2023, *arXiv:2308.12950*.

[8] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, “CodeT5+: Open code large language models for code understanding and generation,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 1069–1088.

[9] R. Pan et al., “Understanding the effectiveness of large language models in code translation,” 2023, *arXiv:2308.03109*.

[10] X. Chen, C. Liu, and D. Song, “Tree-to-tree neural networks for program translation,” in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 2552–2562.

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[12] S. Gao, H. Zhang, C. Gao, and C. Wang, “Keeping pace with ever-increasing data: Towards continual learning of code intelligence models,” in *Proc. 45th Int. Conf. Softw. Eng.*, 2023, pp. 30–42.

[13] GitHub. “The world’s most widely adopted ai developer tool.” 2022. Accessed: Sep. 3, 2024. [Online]. Available: https://github.com/features/copilot

[14] Amazon. “Amazon codewhisperer.” 2022. Accessed: Sep. 3, 2024. [Online]. Available: https://aws.amazon.com/cn/codewhisperer/

[15] Z. AI. “Powerful ai assistant for developers.” 2024. Accessed: Sep. 3, 2024. [Online]. Available: https://codegeex.cn/en-US

[16] GitHub. 2022. Accessed: Sep. 3, 2024. [Online]. Available: https://codesignal.com/resource/developers-and-ai-coding-assistant-trends/

[17] M. L. Siddiq and J. Santos, “Generate and pray: Using SaLLMs to evaluate the security of LLM generated code,” 2023, *arXiv:2311.00889*.

[18] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, “How secure is code generated by ChatGPT?” in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2445–2451.

[19] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, “Lost at C: A user study on the security implications of large language model code assistants,” in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, Anaheim, CA, USA, 2023, pp. 2205–2222.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 754–768.

[21] V. Majdinasab, M. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, “Assessing the security of github copilot’s generated code - a targeted replication study,” in *Proc. IEEE Int. Conf. Softw. Anal., Evol. ReEng. (SANER),* 2024, pp. 435–444.

[22] J. Chen et al., “RMCBench: Benchmarking large language models’ resistance to malicious code,” 2024, *arXiv:2409.15154*.

[23] J. He and M. Vechev, “Large language models for code: Security hardening and adversarial testing,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2023, pp. 1865–1879.

[24] D. Li et al., “Cosec: On-the-fly security hardening of code llms via supervised co-decoding,” in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2024, pp. 1428–1439.

[25] Y. Gu, L. Dong, F. Wei, and M. Huang, “MiniLLM: Knowledge distillation of large language models,” 2023, *arXiv–2306*.

[26] G. Hinton, “Distilling the knowledge in a neural network,” 2015, *arXiv:1503.02531*.

[27] D. Fried et al., “Incoder: A generative model for code infilling and synthesis,” 2022, *arXiv:2204.05999*.

[28] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, and J. Yu, “Security weaknesses of Copilot generated code in GitHub,” 2023, *arXiv:2310.02059*.

[29] R. Agarwal, N. Vieillard, P. Stanczyk, S. Ramos, M. Geist, and O. Bachem, “GKD: Generalized knowledge distillation for auto-regressive sequence models,” 2023, *arXiv:2306.13649*.

[30] Y. Wen, Z. Li, W. Du, and L. Mou, “f-divergence minimization for sequence-level knowledge distillation,” in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, 2023, pp. 10817–10834.

[31] A. Liu et al., “DEXPERTS: Decoding-time controlled text generation with experts and anti-experts,” in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics 11th Int. Joint Conf. Natural Lang. Process. (Volume 1: Long Papers)*, 2021, pp. 6691–6706.

[32] T. Zhong, Q. Wang, J. Han, Y. Zhang, and Z. Mao, “Air-decoding: Attribute distribution reconstruction for decoding-time controllable text generation,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 8233–8248.

[33] L. Qin, S. Welleck, D. Khashabi, and Y. Choi, “Cold decoding: energy-based constrained text generation with Langevin dynamics,” in *Proc. 36th Int. Conf. Neural Inf. Process. Syst.* (*NIPS*), vol. 35, 2024, pp. 9538–9551.

[34] Q. Zheng et al., “CodeGeex: A pre-trained model for code generation with multilingual benchmarking on Humaneval-x,” in *Proc. 29th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, 2023, pp. 5673–5684.

[35] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” 2021, *arXiv:2109.00859*.

[36] A. Lozhkov et al., “StarCoder 2 and the stack v2: The next generation,” 2024, *arXiv:2402.19173*.

[37] GitHub. 2023. GitHub Copilot now has a better ai model and new capabilities. Accessed: Sep. 21, 2024. [Online]. Available: https://github.blog/

[38] J. Wang et al., “Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation,” 2023, *arXiv:2310.16263*.

[39] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, and D. Phung, “VulExplainer: A transformer-based hierarchical distillation for explaining vulnerability types,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 10, pp. 4550–4565, Oct. 2023.

[40] K. Song et al., “Lightpaff: A two-stage distillation framework for pre-training and fine-tuning,” 2020, *arXiv:2004.12817*.

[41] M. Wu, A. Waheed, C. Zhang, M. Abdul-Mageed, and A. F. Aji, “Lamini-LM: A diverse herd of distilled models from large-scale instructions,” 2023, *arXiv:2304.14402*.

[42] N. Ho, L. Schmid, and S.-Y. Yun, “Large language models are reasoning teachers,” in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, 2023, pp. 14852–14882.

[43] Y. Wen, Z. Li, W. Du, and L. Mou, “f-divergence minimization for sequence-level knowledge distillation,” in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Assoc. for Comput. Linguistics, Jul. 2023, pp. 10817–10834. [Online]. Available: https://aclanthology.org/2023.acl-long.605

[44] J. Ko, S. Kim, T. Chen, and S.-Y. Yun, “DistiLLM: Towards streamlined distillation for large language models,” in *Proc. 41st Int. Conf. Mach. Learn.*, 2024, pp. 24872–24895.

[45] Y. Leviathan, M. Kalman, and Y. Matias, “Fast inference from transformers via speculative decoding,” in *Proc. 40th Int. Conf. Mach. Learn.* PMLR, 2023, pp. 19274–19286.

[46] M. Kim, H. Lee, K. M. Yoo, J. Park, H. Lee, and K. Jung, “Critic-guided decoding for controlled text generation,” in *Proc. Findings Assoc. Comput. Linguistics: ACL*, 2023, pp. 4598–4612.

[47] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, “Planning with large language models for code generation,” 2023, *arXiv:2303.05510*.

[48] M. Chen et al., “Evaluating large language models trained on code,” 2021, *arXiv:2107.03374*.

[49] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, “MagiCoder: Empowering code generation with OSS-instruct,” in *Proc. 41st Int. Conf. Mach. Learn.*, 2024, pp. 52632–52657.

[50] E. J. Hu et al., “Lora: Low-rank adaptation of large language models,” 2021, *arXiv:2106.09685*.

[51] L. Gao et al., “The Pile: An 800GB dataset of diverse text for language modeling,” 2020, *arXiv:2101.00027*.

[52] M. Bavarian et al., “Efficient training of language models to fill in the middle,” 2022, *arXiv:2207.14255*

[53] D. Kocetkov et al., “The stack: 3 TB of permissively licensed source code,” 2022, *arXiv:2211.15533*.

[54] B. Hui et al., “QWEN2. 5-coder technical report,” 2024, *arXiv:2409.12186*.

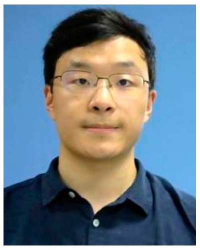
Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.

[55] J. He, M. Vero, G. Krasnopolska, and M. Vechev, “Instruction tuning for secure code generation,” 2024, *arXiv:2402.09497*.
[56] Huggingface. “Hugging face.” 2023. Accessed: Sep. 21, 2024. [Online]. Available: https://huggingface.co/
[57] Z. Gekhman et al., “Does fine-tuning LLMS on new knowledge encourage hallucinations?” 2024, *arXiv:2405.05904*.
[58] D. Guo et al., “DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning,” 2025, *arXiv:2501.12948*.
[59] W. Kwon et al., “Efficient memory management for large language model serving with pagedattention,” in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 611–626.

Dong Li received the M.S. degree from Chongqing Jiaotong University, China. He is currently working toward the Ph.D. degree with the School of Big Data and Software Engineering, Chongqing University, China. His current research interests include LLMs, software security, and AI for software engineering. His work has been published in flagship journals and conferences, including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and ISSTA.

Shanfu Shu received the B.S. degree from Changsha University of Science and Technology. He is currently working toward the M.S. degree with the School of Big Data and Software Engineering, Chongqing University, China. His research interests primarily focus on AI for software engineering and application of LLMs. His work has been published in journals such as *Journal of Software* (JOS).

Meng Yan (Member, IEEE) is currently a Research Professor with the School of Big Data & Software Engineering, Chongqing University, China. His research focuses on how to improve developers' productivity, how to improve software quality, and how to reduce the effort during software development by analyzing rich software repository data. For more information, see https://yanmeng.github.io/.



Zhongxin Liu (Member, IEEE) received the Ph.D. degree in computer science and technology from Zhejiang University, Zhejiang, China, in 2021. He is currently a Distinguished Research Fellow with the College of Computer Science and Technology, Zhejiang University, China. His research interests include AI for software engineering and mining software repositories, especially helping developers to understand, write, and test source code and make informed development decisions by learning from software “Big Data.” For more information, please see https://zhongxin-liu.github.io/.

Chao Liu received the Ph.D. degree in 2018 under the supervision of Prof. Dan Yang from Chongqing University, China. He is currently an Associate Professor with the School of Big Data & Software Engineering, Chongqing University, China. His current research focuses on intelligent software engineering, large language model, code search, code generation, and software visualization.

Xiaohong Zhang received the M.S. degree in applied mathematics and the Ph.D. degree in computer software and theory from Chongqing University, China, in 2006. He is currently a Professor and the Vice Dean of the School of Big Data and Software Engineering, Chongqing University. He has authored over 50 scientific papers and some of them are published in some authoritative journals and conferences, such as IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, IEEE TRANSACTIONS ON IMAGE PROCESSING, PR, PRL, JSS, and IST. His research interests include data mining of software engineering, topic modeling, image semantic analysis, and video analysis.

David Lo (Fellow, IEEE) is a Professor in computer science and the Director of the Information and Systems Cluster with the School of Computing and Information Systems, Singapore Management University. He leads the Software Analytics Research (SOAR) Group. His research interests are in the intersection of software engineering, cybersecurity, and data science, encompassing socio-technical aspects, and analysis of different kinds of software artifacts, with the goal of improving software quality and security and developer productivity.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on December 03,2025 at 02:29:34 UTC from IEEE Xplore. Restrictions apply.