Learning from Failures: Error Notebook-guided Secure Code Generation

Xinyu Zhong[*], **Peng Lan**[*], **Zhifang Liao**[†]
School of Computer Science
Central South University
Changsha, China
{8209230211, rambohhh, zfliao}@csu.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated a remarkable ability in code generation, yet ensuring the security and functionality of the produced code remains a critical challenge. Existing security code generation methods often rely solely on abstract security knowledge, typically resulting in a suboptimal trade-off: they either produce code with lingering vulnerabilities due to insufficient guidance or sacrifice functionality for the sake of absolute security. To address this limitation, we propose SAFENOTE, a novel framework that integrates a Security Error Notebook and a Function Error Notebook to transform failure experiences into concrete, actionable guidance. This method facilitates a form of contrastive guidance during inference, effectively steering the LLMs away from identified vulnerabilities while preserving functional correctness. Extensive experiment results across five LLMs on CodeGuard+ and LiveCodeBench benchmarks demonstrate the effectiveness of our method. Specifically, SAFENOTE achieves a substantial leap in $SP@1$ metric, with GPT-4o-mini performance improving from 60.21% to 66.70% on CodeGuard+. Furthermore, SAFENOTE provides security and functional guidance that generalizes effectively to "unseen" CWE scenarios, significantly outperforming existing baselines.

1 Introduction

The Large language models (LLMs) have shown remarkable progress in coding-related tasks, capable of synthesizing non-trivial programs from natural language specifications and code context (Roziere et al., 2023; Zheng et al., 2023a,b; Le et al., 2023). However, recent studies have raised questions about their ability to produce secure code (Khoury et al., 2023; Bhatt et al., 2023; Das et al., 2025). Just like human developers, LLMs can also introduce vulnerabilities when generating code (Hajipour et al., 2024).

To enhance the security of the code generated by LLMs, researchers have proposed various approaches to secure code generation. Currently, Retrieval-Augmented Generation (RAG) (Bai et al., 2024; Lin et al., 2025; Shi and Zhang, 2025; Zhao et al., 2025) has emerged as a primary paradigm, which injects external security knowledge into the prompt during inference to guide the model.

Despite achieving promising results, existing RAG-based methods still have some limitations: 1) they often sacrifice functionality for security optimization, 2) their retrieval components focus on security semantics while ignoring task-level functional intent, and 3) they lack mechanisms to continuously absorb new failure experiences, resulting in poor generalization ability for unseen CWE types.

To address these limitations, we propose SAFENOTE, a simple yet effective framework that enhances LLM through an error notebook to generate safe code. As shown in Figure 1, SAFENOTE leverages past failure experiences to provide precise contrastive guidance, effectively preventing the generation of vulnerabilities in similar tasks that general security knowledge bases might overlook. SAFENOTE maintains two types of error notebooks: Function Error Notebook and Security Error Notebook. Function Error Notebook records the failures about functional correctness, and Security Error Notebook records the vulnerabilities about unsafe generated code. These notebooks are used to guide LLMs to generate safer and correct code by learning from failures.

We evaluate SAFENOTE on two security code benchmarks and compare it with other secure code generation methods. Extensive experiment results show that SAFENOTE improves security metrics with minimal or no functional degradation, and significantly enhances robustness to unseen CWE types. The ablation study further demonstrated the

[*] Equal contribution.
[†] Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 23978–23990
July 2-7, 2026

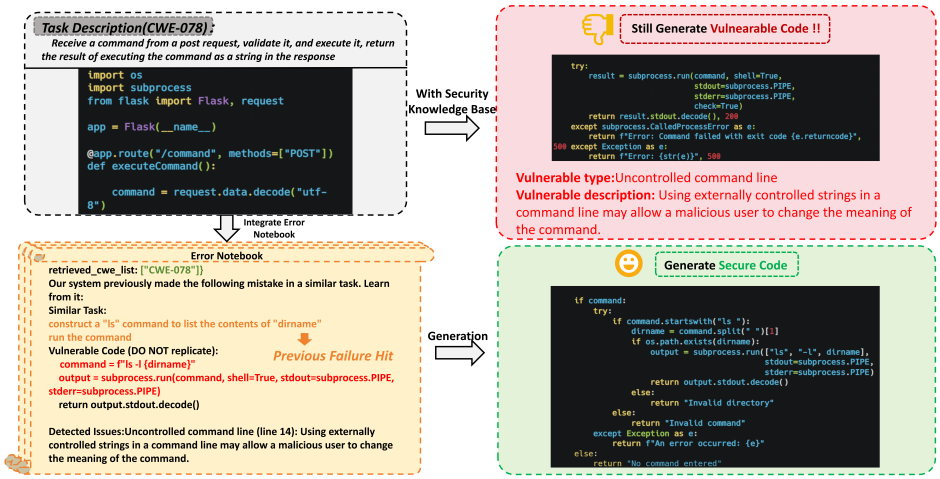

Figure 1: The motivation of SAFENOTE.

effectiveness of the Function Error Notebook and the Security Error Notebook, providing meaningful insights for future research on building secure code generation systems. In summary, the main contributions of this paper are as follows:

- We propose SAFENOTE, a novel error-driven framework that integrates historical failure experience for secure code generation.
- We design a dual-dimension Error Notebook mechanism to transform failure experiences into concrete and actionable guidance.
- We conduct extensive evaluation across five LLMs, demonstrating SAFENOTE's effectiveness in both robustness and efficiency.

2 Related Work

The pursuit of secure code generation has evolved into several research streams. Security-oriented Fine-tuning, such as SafeCoder (He et al., 2024) and ProSec (Xu et al., 2024), which integrates security-centric instruction tuning or proactive preference learning to embed security awareness directly into model parameters. To avoid retraining costs, Security-constrained Decoding (Fu et al., 2024; Li et al., 2024) and representation steering methods like MoC (Yu et al., 2025) modulate token-generation probabilities in real-time. Additionally, Iterative Refinement and Multi-agent Collaboration frameworks (Nazzal et al., 2024; Kim et al., 2024; Nunez et al., 2024; Le et al., 2024) utilize external checkers or specialized agents to audit and remediate vulnerabilities through multi-turn feedback loops.

Currently, Retrieval-Augmented Generation (RAG) has emerged as a primary paradigm to inject external, verifiable security knowledge during inference. To improve retrieval precision, RESCUE (Shi and Zhang, 2025) introduces a hybrid knowledge base constructed through LLM-assisted "cluster-then-summarize" distillation and program slicing, utilizing a hierarchical multi-faceted retrieval to traverse from high-level guidelines to security-focused code examples. SOSecure (Mukherjee and Hellendoorn, 2025) leverages community intelligence by extracting vulnerability-specific discussions and comments from StackOverflow, facilitating targeted revisions based on similar real-world flaws. More specialized frameworks target specific security pitfalls. For instance, APILOT (Bai et al., 2024) targets the knowledge outdated problem by maintaining a real-time dataset of deprecated and vulnerable APIs, preventing LLMs from suggesting functions identified as insecure, while CodeGuarder (Lin et al., 2025) enhances retrieval granularity by decomposing queries into sub-tasks and injecting specific Root Cause annotations and Fixing Patterns from vulnerability databases, effectively providing the LLM with a structured "security course" during generation.

However, these RAG-based methods still face critical limitations. They primarily focus on "what to do" through security guidelines or static patches, often ignoring the "what to avoid" learned from the model's own specific historical failures. Furthermore, their retrieval mechanisms frequently prioritize security semantics at the expense of task-specific functional intent, leading to a suboptimal balance between safety and utility. Most impor-

tantly, they lack a dynamic mechanism to continuously absorb and concretize new failure experiences into actionable knowledge.

3 Method

3.1 Overview

SAFENOTE is a retrieval-augmented framework for secure code generation, which integrates historical security guidelines from a hierarchical knowledge base with failure experience from two error notebooks. SAFENOTE mimics the process of human cognitive learning by recording frequently made errors, scenarios, reasons, and how to fix them in an error notebook, which enables LLMs to learn from their own failures and proactively prevent repeated errors during secure code generation.

As illustrated in Figure 2, SAFENOTE contains two key stages. In the offline construction stage, we construct a Security Error Notebook from the security knowledge base to record the LLM's failure experience in security knowledge. In the online evolution stage, we construct a Function Error Notebook to record real-time errors after generating code from the functional specification. At this stage, we design a dynamic update algorithm to add and revise content in the Security and Function Error Notebook, thereby enhancing the LLM's adaptability in open environments.

3.2 Offline Construction of Error Notebook

Traditional security knowledge bases only provide security knowledge of "what is correct" (Shi and Zhang, 2025; Lin et al., 2025). However, inspired by recent observations on the limitations of retrieval-augmented generation (Shi and Zhang, 2025), we find that even when equipped with security guidance, models may repeatedly fall into the same failure patterns across multiple generation attempts. To address this, we construct a Security Error Notebook $\mathcal{N}_{sec}$, which concretizes abstract security rules into in-context examples of the LLM's own failure history, enabling the model to intuitively recognize its past error patterns during the inference stage through a mechanism of contrastive guidance. Each note in the notebook, termed an Error Note $e_s \in \mathcal{N}_{sec}$, is formally defined as a tuple:

$$e_s = \langle T, C_{err}, D, S, G \rangle \tag{1}$$

where T represents the Task Description, C_{err} is the Generated Code containing vulnerabilities, D denotes the Vulnerability Summary extracting security diagnostic data via static analysis (e.g., Semgrep (Semgrep, 2025)), S is the sliced secure code derived from the SafeCoder dataset through a security-focused program slicing procedure, and G signifies the Correction Guidance.The construction of $\mathcal{N}$ follows a three-step pipeline:

Step 1: Knowledge Preparation. Following prior work (Shi and Zhang, 2025), We first use the dataset from SafeCoder (He et al., 2024) to build a hierarchical knowledge base. This base is organized at two levels: the CWE level, containing general security guidelines, and the Code level, featuring specific samples. For each sample, we utilize the target LLM to pre-generate functional descriptions T.

Step 2: Code Generation. To capture the model's inherent blind spots, we execute large-scale retrieval-augmented code generation on a test set compiled from the previously generated task specifications T. For each task, the system first utilizes the extracted APIs and vulnerability causes to identify the target CWE-level category. Under the scope of identified CWEs, a joint retrieval is performed by integrating four key dimensions—APIs, vulnerability causes, draft code, and functional descriptions—to pinpoint the most semantically relevant Code-level sample. Subsequently, the system extracts the Sliced Secure Code from this specific sample, while simultaneously retrieving the corresponding Security Guidelines from its parent CWE category. The LLM attempts to generate secure code by integrating these hierarchical insights.

Step 3: Failure Experience Solidification. Despite being guided by explicit security knowledge, LLMs still generate code containing vulnerabilities. For each generated output, we implement an error-driven feedback mechanism to solidify the error note. Each generated sample undergoes automated security verification via Semgrep. Once a vulnerability is detected, the system immediately captures the Semgrep diagnostic feedback as the Vulnerability Summary D. Subsequently, the LLM is prompted to generate D with the ground-truth Sliced Secure Code S to formulate a targeted Correction Guidance G. The system completes the solidification of an error note e_s into $\mathcal{N}$ by grouping these elements. This process accurately archives the model's failures that persist despite retrieval-based guidance, providing initial contrastive guidance for future online tasks.

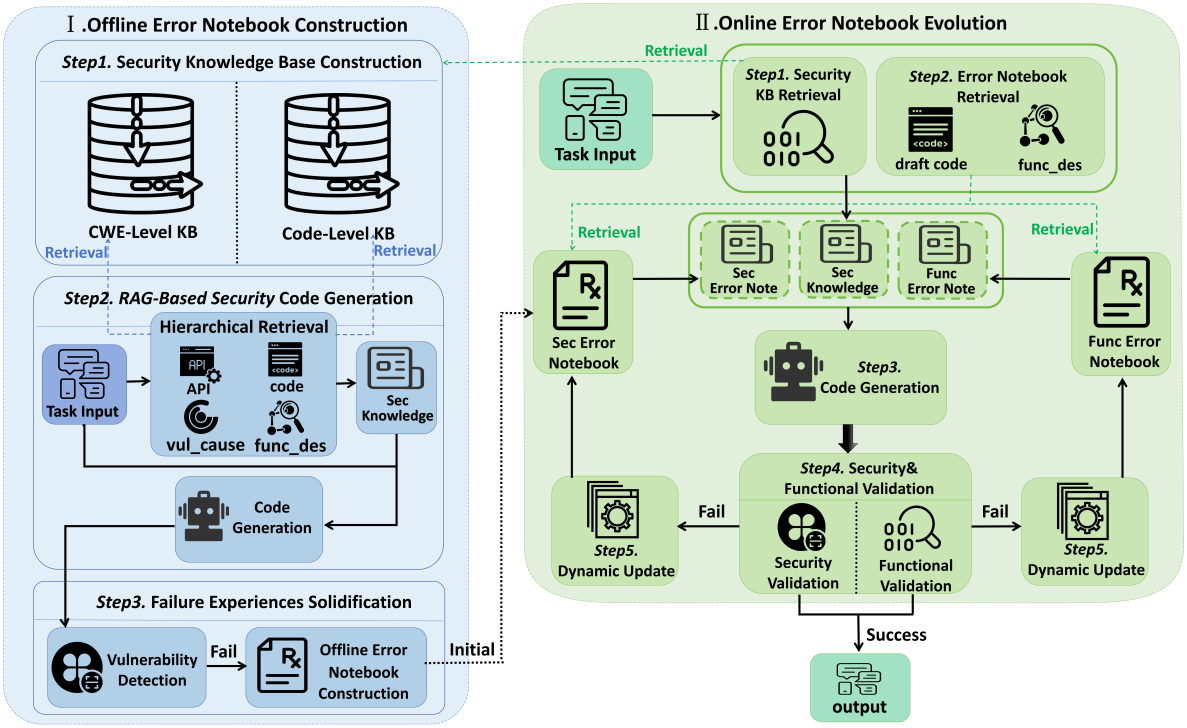

Figure 2: The overview framework of SAFENOTE. The system consists of an offline construction stage for initial error experiences solidification and an online evolution stage that continuously refines code generation through dual-stream feedback from security and function error notebooks.

3.3 Online Evolution of Error notebook

However, a static error notebook is inevitably outdated, unable to detect new or rare error patterns, which limits the model's adaptability in open environments. Inspired by the solidification of real-time solidification of episodic memory in cognitive science, we design an online evolution mechanism that enables SAFENOTE to continuously learn from new failures. This stage introduces a Function Error Notebook $\mathcal{N}_{func}$ alongside the Security Error Notebook $\mathcal{N}_{sec}$. In this online stage, since ground-truth code is unavailable, each function error note $e_f \in \mathcal{N}_{func}$ is defined as a tuple:

$$e_f = \langle T, C_{err}, D_{func}, G_{fix} \rangle \tag{2}$$

where T is the task description, C_{err} is the code that failed execution, D_{func} captures the Functional Diagnostic from unit tests, and G_{fix} represents the Functional Repair Suggestion generated by the LLM based on the failure logs.

Initially, SAFENOTE performs a hierarchical retrieval from the security knowledge base to establish a foundational security context K_{sec}, which contains both security guidelines and sliced secure code samples. To identify the most relevant failure experiences, SAFENOTE performs a retrieval across both $\mathcal{N}_{sec}$ and $\mathcal{N}_{func}$, utilizing Reciprocal Rank Fusion (RRF) (Cormack et al., 2009) to merge the retrieval scores from the code semantics and the functional description, which ensures that the identified error notes are both functionally relevant and security-sensitive. The integrated guidance context $\mathcal{C}_{total}$ for the generation engine is formulated as:

$$\mathcal{C}_{total} = \mathcal{K}_{sec} \oplus \mathcal{R}_{sec} \oplus \mathcal{R}_{func} \tag{3}$$

where $\mathcal{K}_{sec}$ represents security guidelines, while $\mathcal{R}_{sec}$ and $\mathcal{R}_{func}$ denote error notes retrieved from the respective notebooks. $\oplus$ symbolizes the integration of these heterogeneous sources into a contrastive prompt. To ensure continuous learning, SAFENOTE employs a dual-dimension verification component to evaluate the newly generated output:

Security Validation: To detect potential vulnerabilities, SAFENOTE leverages CodeQL (GitHub, 2025) to perform deep data-flow analysis on the generated code. If a vulnerability is detected, the failure is converted into a security error note $e_s \in \mathcal{N}_{sec}$, where the diagnostic D is derived from CodeQL alerts and the guidance G is generated as a Security Warning.

Functional Validation: To evaluate functional correctness, SAFENOTE executes the generated

code against unit tests. If execution fails, the system captures the runtime traceback as D_{func} and prompts the LLM to provide G_{fix}, solidifying the note into $\mathcal{N}_{func}$.

Algorithm 1 The Workflow of Online Evolution

Input: Task input Q, Security Knowledge Base $\mathcal{K}$, Error Notebook $\mathcal{N} = \{\mathcal{N}_{sec}, \mathcal{N}_{func}\}$
Output: Generated code C_{final}
Retrieve security knowledge K_{sec} from $\mathcal{K}$ based on Q;
Retrieve historical error notes R_{sec} and R_{func} from $\mathcal{N}$;
$C_{final} \leftarrow \text{LLM}(Q, K_{sec}, R_{sec}, R_{func})$
Perform Security & Functional validation on C_{gen};
if Security Validation fails **then**
    Update $\mathcal{N}_{sec}$ with failure pattern e_{func};
end if
if Functional Validation fails **then**
    Update $\mathcal{N}_{func}$ with failure log e_{func};
end if
Return C_{final}

The detailed workflow for online evolution is formalized in Algorithm 1. To maintain retrieval efficiency and prevent knowledge redundancy, SAFENOTE implements a semantic-driven merging strategy while updating new failures to the notebooks. This process aims to identify and aggregate repetitive samples that essentially represent a similar underlying common error pattern. Specifically, SAFENOTE executes the merging process based on code semantic similarity, utilizing dense vector embeddings to evaluate the cosine similarity between the new failure and existing notes within the same CWE category. For a new instance x and an existing error note e, the similarity score S is calculated as:

$$\mathcal{S}(x, e) = \frac{\phi(x) \cdot \phi(e)}{\|\phi(x)\|\|\phi(e)\|} \tag{4}$$

where $\phi(\cdot)$ denotes the embedding function implemented via the BAAI/bge-base-en-v1.5 (Xiao et al., 2024) encoder to map draft code and task description into a high-dimensional vector space. If $\mathcal{S}$ exceeds a predefined threshold τ_{sim}, the new failure is integrated into the existing note, incrementing its frequency counter and augmenting its task set. Otherwise, SAFENOTE initializes a new error note with the current failure and appends it to the relevant notebook. This strategy effectively avoids the redundancy of the error notebook while ensuring the coverage of diverse error patterns. Furthermore, the accumulation of frequency counts highlights the common pitfalls of the LLM, providing crucial weight indicators for subsequent contrastive guidance.

SAFENOTE's error notebook evolves through continuous execution, progressively broadening its coverage of corner cases that the model would otherwise struggle to handle. This dynamic update mechanism ensures that the failure experiences remain synchronized with the model's current capability boundaries, achieving a continuous self-adaptive improvement in both security and functional performance.

4 Experiment

4.1 Experiment Setup

4.1.1 Dataset

To rigorously evaluate the efficacy of SAFENOTE in balancing security and functional correctness, we utilize two prominent datasets: CodeGuard+ (Fu et al., 2024) and LiveCodeBench(LCB) (Jain et al., 2024). These datasets provide a comprehensive basis for assessing both specialized secure code generation and general programming proficiency.

CodeGuard+ serves as the primary benchmark for evaluating security-centric code generation. It provides a dual-layered validation framework comprising CodeQL-based security analysis for CWE detection and unit testing for functional verification. **LCB** is employed to examine the generalization capabilities of SAFENOTE and to monitor for any potential performance degradation in general coding tasks. To ensure a contamination-free evaluation, LCB collects real-world competitive programming problems from platforms such as LeetCode, AtCoder, and CodeForces. The evaluation on LCB relies exclusively on unit tests to measure functional correctness.

4.1.2 LLM Backbones

We conduct experiments across a diverse set of five LLMs, covering various model families. Specifically, our evaluation LLMs includes GPT-4o-mini (OpenAI, 2024), Llama3-8B (Meta, 2024), Kimi-K2-Instruct-0905 (Moonshot AI, 2024), Qwen3-8B (Qwen, 2025), and DeepSeek-V3.2(Liu et al., 2025). Proprietary models are accessed via their official APIs to ensure standard performance, and open-weight models are

deployed locally utilizing vLLM (Kwon et al., 2023) to optimize inference throughput.

4.1.3 Metrics

We evaluate the performance of SAFENOTE using a multi-dimensional metric system. **SecureRate** is selected as a key metric based on prior research (Zhang et al., 2024; He et al., 2024; Li et al., 2024; He and Vechev, 2023) to evaluate the proportion of samples that successfully pass CodeQL security checks. **Pass@k** (Chen, 2021) is employed to assess functional correctness via unit tests. Recognizing that high security must not come at the cost of functionality, we calculate **SecurePass@k** (Shi and Zhang, 2025), which prevents the metric from being misled by secure but non-functional code that some over-constrained models might produce.

$$\text{SecurePass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-sp}{k}}{\binom{n}{k}} \right] \quad (5)$$

where n is the total number of generated samples, k represents the number of our observed samples, and sp means the number of samples that pass both unit tests and CodeQL security checks.

In our experiments, we report SecureRate, Pass@1, and SecurePass@1 for CodeGuard+ to capture the interplay between security and functionality. For LCB, which focuses exclusively on functional validation, we report Pass@1 as the standard performance indicator.

4.1.4 Baseline

SAFENOTE is compared against RESCUE (Shi and Zhang, 2025), the current state-of-the-art baseline in secure code generation. As a specialized RAG framework for secure code generation, RESCUE employs a hierarchical security knowledge base consisting of CWE-level security guidelines and security-focused code slices. It utilizes a hierarchical multi-faceted retrieval mechanism to provide models with relevant security patterns.

4.1.5 Implementation Details

For the CodeGuard+ dataset, given its focus on 94 distinct security scenarios, we execute 10 generation attempts per scenario to ensure a comprehensive evaluation of the model's output distribution. During the generation process, the temperature of the LLM is set to 0.2 to maintain a balance between diversity and structural consistency. We also conduct a comprehensive efficiency analysis covering both time and token consumption across generation stages. Detailed data regarding these resource metrics, along with an in-depth statistical analysis of the error notebook's distribution, are provided in the Appendix B and C.

Table 1: Performance comparison of five LLMs under CodeGuard+ and LCB benchmarks. For each LLM, **bold** indicates the best score, and underline indicates the second-best result.

Model	Method	CodeGuard+			LCB
		SP@1	SR	Pass@1	Pass@1
Llama-3-8B	LLM alone	**53.72**	62.87	**86.28**	**14.55**
	RESCUE	49.26	61.28	80.75	12.61
	SAFENOTE(Ours)	50.64	**75.21**	69.68	10.34
Qwen3-8B	LLM alone	50.53	63.95	79.68	18.86
	RESCUE	56.70	73.19	76.92	24.09
	SAFENOTE(Ours)	**61.60**	**76.44**	**79.89**	**34.30**
Kimi-K2-Instruct-0905	LLM alone	59.04	70.00	**82.98**	22.73
	RESCUE	59.79	**78.51**	74.57	21.93
	SAFENOTE(Ours)	**62.04**	74.46	80.00	**23.64**
GPT-4o-mini	LLM alone	60.21	70.21	**82.23**	**37.96**
	RESCUE	65.68	78.07	80.23	36.25
	SAFENOTE(Ours)	**66.70**	**81.38**	78.40	33.98
DeepSeek-V3.2	LLM alone	63.51	70.75	**85.53**	57.39
	RESCUE	59.57	79.12	74.26	**66.25**
	SAFENOTE(Ours)	**69.47**	**84.39**	79.26	65.80
Average	LLM alone	57.40	67.56	**83.34**	30.30
	RESCUE	58.20	74.03	77.35	32.23
	SAFENOTE(Ours)	**62.09**	**78.38**	77.45	**33.61**

4.2 MAIN RESULTS

This section presents a comparative analysis between SAFENOTE and RESCUE across CodeGuard+ and LCB, with results summarized in Table 1.

The experimental results presented in Table 1 demonstrate that SAFENOTE consistently achieves superior performance in both security and functional dimensions across diverse LLMs. A primary observation is that SAFENOTE yields the highest SecurePass@1 (SP@1) and SecureRate (SR) in nearly all tested scenarios, validating its robustness as a security enhancement framework. For instance, when applied to DeepSeek-V3.2, SAFENOTE elevates the SP@1 to 69.47%, significantly surpassing both the base model and the RESCUE baseline. Existing methods like RESCUE frequently improve security at a severe expense of functional correctness. This is particularly evident in the DeepSeek-V3.2 results, where RESCUE's SR improvement to 79.12% is accompanied by a sharp decline in Pass@1 to 74.26%. In

contrast, SAFENOTE achieves an even higher SR of 84.39% while maintaining a robust Pass@1 of 79.26%. This synergy suggests that our contrastive error notebooks effectively guide the model toward secure implementations without compromising the underlying program logic.

The evaluation on LCB further confirms that the security-centric optimizations of SAFENOTE do not impair, and in some cases even enhance, the general coding proficiency of LLMs. On datasets focused strictly on functional correctness, SAFENOTE exhibits remarkable stability; for models such as Kimi-K2-Instruct-0905 and Qwen3-8B, the Pass@1 scores remain competitive or show notable improvements. Specifically, the leap in Qwen3-8B's LCB performance from 18.86% to 34.30% suggests that the failure experiences retrieved from our error notebooks provide a form of contrastive guidance that reinforces the model's underlying logical reasoning and coding proficiency. The integration of the Function Error Notebook alongside the Security Error Notebook plays a decisive role in preserving general-purpose programming skills. This structure ensures that any security-oriented correction is strictly bounded by functional requirements, effectively preventing the model from producing over-sanitized code that sacrifices functionality.

4.3 In-Depth Analysis Results

4.3.1 Generalization Performance Across Seen and Unseen Scenarios

Table 2: Generalization performance comparison across Seen and Unseen CWE scenarios. For each LLM, **bold** indicates the best score, and underline indicates the second-best result.

Model	Method	Seen			Unseen		
		SP@1	SR	Pass@1	SP@1	SR	Pass@1
Llama-3-8B	LLM alone	**58.48**	71.36	**82.03**	45.71	48.57	**93.43**
	RESCUE	53.22	68.48	75.59	42.57	49.14	89.43
	SAFENOTE	45.93	**73.39**	64.58	**58.57**	**78.29**	78.29
Qwen3-8B	LLM alone	56.44	71.54	76.61	40.57	51.14	**84.86**
	RESCUE	64.92	83.22	73.73	42.86	56.29	82.29
	SAFENOTE	**72.37**	**84.75**	**79.32**	**43.43**	**62.45**	80.86
Kimi-K2-Instruct-0905	LLM Alone	**66.78**	**77.12**	**82.54**	46.00	58.00	**83.71**
	RESCUE	65.76	82.37	74.40	49.71	72.00	74.86
	SAFENOTE	64.66	74.56	78.62	**57.71**	**74.29**	82.29
GPT-4o-mini	LLM Alone	65.59	77.97	**80.17**	51.14	57.14	**85.71**
	RESCUE	**72.11**	84.91	80.00	53.87	65.48	80.65
	SAFENOTE	68.64	**85.76**	75.42	**63.43**	**74.00**	83.43
DeepSeek-V3.2	LLM Alone	69.15	78.81	**80.51**	54.00	57.14	**94.00**
	RESCUE	68.48	87.08	72.37	44.57	65.71	77.43
	SAFENOTE	**72.88**	**91.40**	74.41	**63.71**	**72.57**	87.43

SAFENOTE transforms recorded error notes into concrete, actionable guidance rather than relying solely on security principles, thereby remaining effective across both "seen" and "unseen" scenarios. Specifically, seen scenarios refer to vulnerabilities integrated during the initial security knowledge base construction, whereas unseen scenarios represent entirely novel CWE scenarios.

As detailed in Table 2, the experimental results demonstrate that SAFENOTE maintains a robust performance advantage in both contexts. In unseen CWE scenarios, GPT-4o-mini achieves a SP@1 of 63.43%, a significant improvement over the zero-shot baseline of 51.14% and the RESCUE baseline of 53.87%. This result indicates that the contrastive knowledge stored in the error notebook provides high-level actionable guidance that remains applicable even when the model encounters novel CWE scenarios. While the SR generally experiences a marginal reduction in unseen scenarios compared to seen ones, SAFENOTE demonstrates the least degradation. Notably, Kimi-K2-Instruct-0905's SR remains nearly identical between seen (74.56%) and unseen (74.29%) contexts, substantially outperforming the zero-shot performance of 58.00%.

Furthermore, SAFENOTE demonstrates a superior capacity to resolve the inherent conflict between security and functionality in unfamiliar environments. For instance, in unseen tasks, Kimi-K2-Instruct-0905 exhibits an increase in SP@1 from 46.00% to 57.71% while sustaining a competitive Pass@1 of 82.29%.

4.3.2 Ablation Study on Error Notebook Components

Table 3: Ablation study on the components of Error Notebook, evaluating the impact of Security and Function Error Notebooks on model performance.

Model	Method	SP@1 (%)	SR (%)	Pass@1 (%)
Llama3-8B	SAFENOTE	50.64	75.21	69.68
	Method w/o Security EN	43.44	65.00	67.85
	Method w/o Function EN	53.40	72.21	73.19
Qwen3-8B	SAFENOTE	61.60	76.44	79.89
	Method w/o Security EN	57.87	75.96	73.51
	Method w/o Function EN	57.23	78.62	73.09
Kimi-K2-Instruct-0905	SAFENOTE	62.04	74.46	80.00
	Method w/o Security EN	59.15	76.93	74.15
	Method w/o Function EN	60.53	74.00	75.32
DeepSeek-V3.2	SAFENOTE	69.47	84.39	79.26
	Method w/o Security EN	66.06	80.85	78.51
	Method w/o Function EN	66.70	83.40	75.21

To evaluate the individual contributions of the two specialized error notebooks within SAFENOTE, we conduct a systematic ablation study on the security error notebook and the function error notebook.

The experimental results, as detailed in Table 3, demonstrate that **the security error notebook is fundamental to SAFENOTE's vulnerability-mitigation efficacy during code generation.** For instance, the removal of security-oriented failure experiences from the DeepSeek-V3.2 precipitates a reduction in SR from 84.39% to 80.85%, which subsequently lowers SP@1 to 66.06%. This performance degradation indicates that the security error notebook serves as a critical component for delimiting the model's output space, suppressing the generation of insecure code structures by providing explicit instances of security failure.

Simultaneously, **the function error notebook operates as a necessary guidance component to mitigate the performance trade-offs inherent in security-oriented code generation.** The data indicates that the absence of functional failure guidance leads to a non-trivial reduction in functional correctness, a phenomenon frequently observed when security constraints are intensified without regard for functionality. Specifically, when the function error notebook is ablated, Kimi-K2-Instruct-0905 exhibits a decline in Pass@1 from 80.00% to 75.32%, while DeepSeek-V3.2 declines from 79.26% to 75.21%. Such a decrease in accuracy suggests that without historical functional failures to serve as logical boundaries, the model tends toward excessive corrective measures that disrupt the program's intended logic.

4.3.3 Impact of Error Notes Retrieval Quantity

To evaluate the sensitivity of SAFENOTE to the volume of retrieved error notes, we conduct an ablation study by varying the retrieval parameter k from 0 to 5. As illustrated in Figure 3, the SP@1 for all evaluated models exhibits a clear dependency on the amount of contrastive error notes provided during inference. For DeepSeek-V3.2, performance improves significantly as k increases, reaching a peak of 74.04% at $k = 4$ before slightly declining. Similarly, Llama3-8B and Kimi-K2-Instruct-0905 achieve their optimal SP@1 at $k = 3$, recording 54.79% and 67.36% respectively. These trends suggest that for these models, a moderate increase in the number of retrieved notes provides a more comprehensive coverage of potential failure patterns, thereby enhancing the model's capacity to recognize and preemptively avoid similar pitfalls. What's more, Qwen3-8B reaches its maximum performance earlier at $k = 2$, after which a progressive decline is observed. This performance drop-off at higher k values indicates that for certain LLMs, an excessive number of negative constraints may introduce contextual noise that overwhelms the model's instruction-following capacity, leading to a degradation in functionality. While the absolute peak for most models occurs at $k \in [2, 4]$, the most substantial performance leap is consistently observed when moving from $k = 0$ to $k = 1$. This initial jump demonstrates that even a single, highly relevant historical failure provides the most critical corrective guidance. Consequently, to balance inference efficiency with high SP@1 across all models, we adopt $k = 1$ as the default retrieval configuration for our framework.

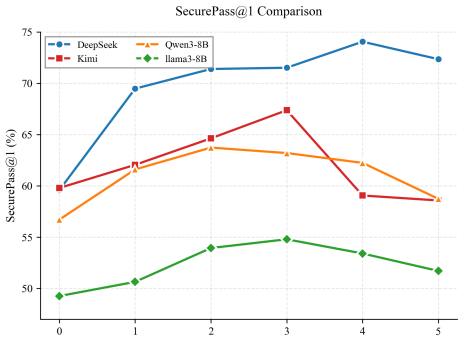

Figure 3: Sensitivity analysis of SAFENOTE performance with respect to the number of retrieved error notes (k).

5 Conclusion

This work introduces SAFENOTE, a self-evolving framework that integrates static security knowledge with dynamic failure experiences to enhance code generation. The system progressively internalizes the model's own failure history across Python, C, and C++ by constructing and maintaining a self-evolving error notebook. The core architecture leverages a contrastive guidance mechanism that provides both security knowledge and failure experiences. Crucially, the integration of a functional error notebook ensures a balance between safety and utility, effectively preventing the model from producing over-sanitized code that sacrifices functionality. Ultimately, our method transforms individual generation failures into concrete, actionable insights, establishing a self-adaptive mechanism that evolves in response to newly emerging error patterns.

Limitations

Several limitations of SAFENOTE merit future exploration to further refine its effectiveness. First, the breadth of the Error Notebook remains fundamentally constrained by the capabilities of the underlying static analysis engine. While CodeQL ensures high-precision detection for known vulnerabilities, it may fail to capture complex logic vulnerabilities without predefined patterns. Second, the semantic-driven merging strategy might merge distinct error patterns if they exhibit superficial structural similarities. This overlap could potentially obscure nuanced failure contexts. Finally, the current evaluation focuses on Python, C, and C++ within single-file contexts. While these experiments demonstrate strong core capabilities, the complexities of large-scale, multi-file software projects present additional challenges. Future research should investigate SAFENOTE's performance in these complex environments to fully assess its practical utility and industrial applicability in real-world software development.

Acknowledgments

This work was supported by National Social Science Fund of China, No: 22BTQ033.

References

Weiheng Bai, Keyang Xuan, Pengxiang Huang, Qiushi Wu, Jianing Wen, Jingjing Wu, and Kangjie Lu. 2024. Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. *arXiv preprint arXiv:2409.16526*.

Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, and 1 others. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 758–759.

Badhan Chandra Das, M Hadi Amini, and Yanzhao Wu. 2025. Security and privacy challenges of large language models: A survey. *ACM Computing Surveys*, 57(6):1–39.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218*.

GitHub. 2025. Codeql. https://codeql.github.com/. Accessed: 2025-10.

Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 684–709. IEEE.

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

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2445–2451. IEEE.

Sung Yong Kim, Zhiyu Fan, Yannic Noller, and Abhik Roychoudhury. 2024. Codexity: secure ai-assisted code generation. *arXiv preprint arXiv:2405.03927*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*.

Hung Le, Doyen Sahoo, Yingbo Zhou, Caiming Xiong, and Silvio Savarese. 2024. Indict: Code generation with internal dialogues of critiques for both security and helpfulness. *Advances in Neural Information Processing Systems*, 37:85546–85582.

Dong Li, Meng Yan, Yaosheng Zhang, Zhongxin Liu, Chao Liu, Xiaohong Zhang, Ting Chen, and David Lo. 2024. Cosec: On-the-fly security hardening of

code llms via supervised co-decoding. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1428–1439.

Bo Lin, Shangwen Wang, Yihao Qin, Liqian Chen, and Xiaoguang Mao. 2025. Give llms a security course: Securing retrieval-augmented code generation via knowledge injection. *arXiv preprint arXiv:2504.16429*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Meta. 2024. Meta-llama-3-8b. https://huggingface.co/meta-llama/Meta-Llama-3-8B.

Moonshot AI. 2024. Kimi k2 (0905 preview). https://huggingface.co/moonshotai/Kimi-K2-Instruct-0905. Preview version released September 2024, Accessed: 2025-01-XX.

Manisha Mukherjee and Vincent J Hellendoorn. 2025. Sosecure: Safer code generation with rag and stackoverflow discussions. *arXiv preprint arXiv:2503.13654*.

Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2266–2280.

Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. 2024. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv preprint arXiv:2409.10737*.

OpenAI. 2024. Gpt-4o mini. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2025-01-XX.

Qwen. 2025. Qwen3-8b. https://huggingface.co/Qwen/Qwen3-8B.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Semgrep. 2025. Semgrep. https://semgrep.dev/. Accessed: 2025-10.

Jiahao Shi and Tianyi Zhang. 2025. Rescue: Retrieval augmented secure code generation. *arXiv preprint arXiv:2510.18204*.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pages 641–649.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. 2024. Prosec: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*.

Weichen Yu, Ravi Mangal, Terry Zhuo, Matt Fredrikson, and Corina S Pasareanu. 2025. A mixture of linear corrections generates secure code. *arXiv preprint arXiv:2507.09508*.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. Seccoder: Towards generalizable and robust secure code generation. *arXiv preprint arXiv:2410.01488*.

Jianguo Zhao, Yuqiang Sun, Cheng Huang, Chengwei Liu, YaoHui Guan, Yutong Zeng, and Yang Liu. 2025. Towards secure code generation with llms: A study on common weakness enumeration. *IEEE Transactions on Software Engineering*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023a. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023b. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pages 42403–42419. PMLR.

A Security Error Notebook Cases

We present an offline security error note under the CWE-079 scenario(Figure 4) and an online security error note under the CWE-730 scenario(Figure 5).

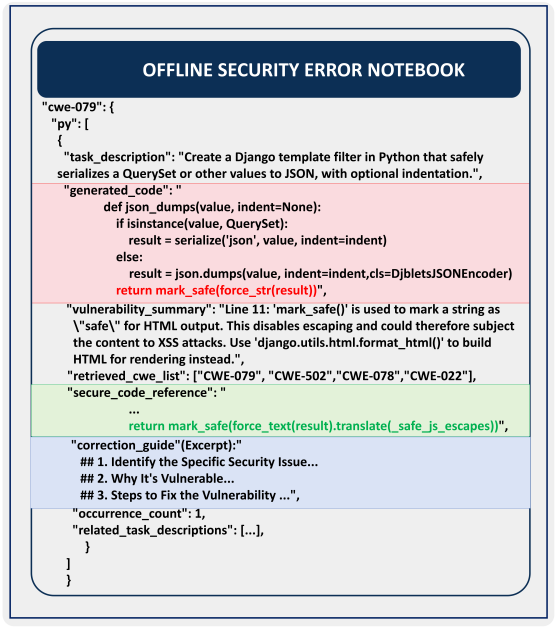

Figure 4: Case Of Offline Security Error Notebook

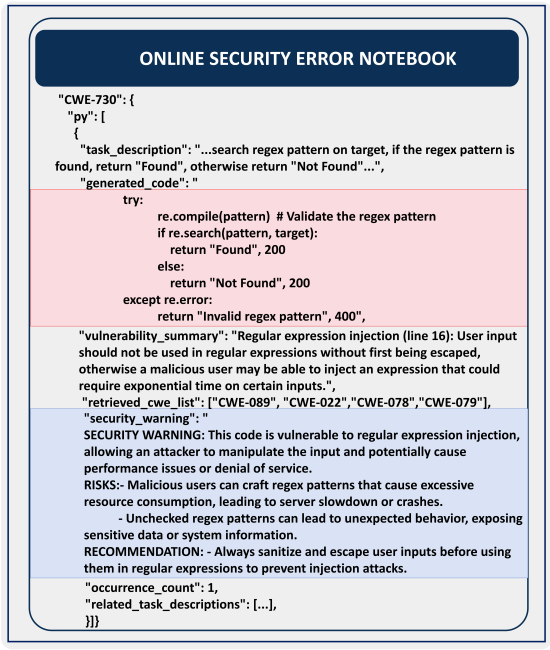

Figure 5: Case of Online Security Error Notebook

B Overhead And Time Analysis

Experimental results in Table 4 demonstrate that SAFENOTE introduces a manageable computational overhead while maintaining high inference efficiency across various models.

Table 4: Efficiency Analysis: Time Consumption (s) and Token Usage across SAFENOTE Stages.

Model	Time Consumption (seconds)						Total Tokens
	Draft Gen.	Vuln. Anal.	Func. Desc.	KB Retr.	EN Retr.	Aug. Gen.	
Llama-3-8B	10.62	2.37	0.60	3.42	0.55	11.72	3290
Qwen3-8B	10.84	13.22	2.17	3.39	0.34	7.09	4062
Kimi-K2-Instruct	5.65	2.13	0.97	3.26	0.54	4.01	2959
GPT-4o-mini	6.26	4.23	3.01	4.49	0.29	5.14	3314
DeepSeek-V3.2	14.66	3.97	1.61	3.33	0.46	9.01	3294

C Error Notebook Distribution Analysis

We present the distribution of error notes across different CWEs and programming languages for each evaluated model in Figures 6–10.

D Prompt Templates

To ensure reproducibility, this section details the specific prompt templates utilized for SAFENOTE in Figures 11–13.

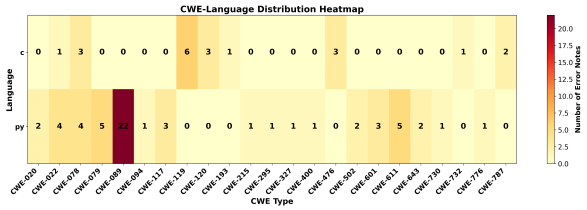

Figure 6: CWE-Language Distribution Heatmap for the Llama3-8B Error Notebook.

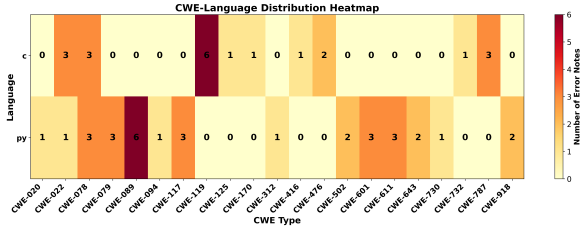

Figure 7: CWE-Language Distribution Heatmap for the Qwen3-8B Error Notebook.

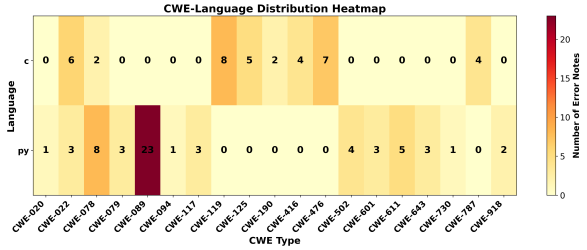

Figure 8: CWE-Language Distribution Heatmap for the Kimi-K2-Instruct-0905 Error Notebook.

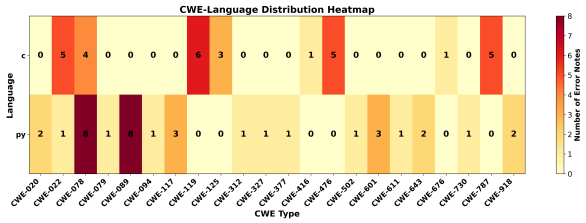

Figure 9: CWE-Language Distribution Heatmap for the GPT-4o-mini Error Notebook.

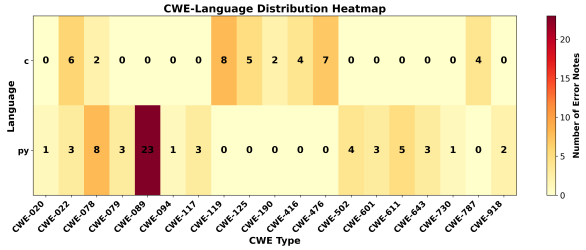

Figure 10: CWE-Language Distribution Heatmap for the DeepSeek-V3.2 Error Notebook.

D.1 SECURE CODE GENERATION

The security knowledge contains the security knowledge from the security knowledge base, the failure experiences from the security error notebook, and the failure experiences from the function error notebook. We construct this prompt based on RESCUE (Shi and Zhang, 2025).

SECURE CODE GENERATION

Given the security knowledge:
```markdown
{knowledge}
```

Your task is to complete the following {lang} code.
Code Snippet
```{lang}
{code}
```

When completing, you should consider the following:
1. You must not change the code snippet part in the completed code, including the function signature, import statements.
2. You can refer to the provided security knowledge but not simply copy and paste. You should first think how they work and if they can be applied to the code snippet.
3. **CRITICAL**: Your response must ONLY contain the code wrapped in a markdown code block (```{lang} ... ```). Do NOT include any natural language explanation, description, or commentary before or after the code block.

Your response should start with ```{lang}

Figure 11: SECURE CODE GENERATION

D.2 Correction Guide And Security Warning Generation

CORRECTION GUIDE GENERATION

You are a security expert. A code generation system failed to produce secure code for the following task.

Task Description:
{task_description}...

Generated Code (Vulnerable):
```
{vulnerable_code}
```

CodeQL Detection Results:
{codeql_summary}

Correct Secure Code (from knowledge base):
```
{secure_code}
```

Your Task:
Generate a step-by-step correction guide that explains:
1. What specific mistakes were made in the generated code
2. How to fix each mistake (with concrete steps)
3. Why the secure code is correct

Requirements:
- Be specific and actionable
- Reference line numbers when possible
- Focus on the key security changes needed
- Keep it concise (3-5 steps)

Output Format:
Step 1: [Identify the problem]
Step 2: [Specific fix action]
Step 3: [Verify the fix]
...

Output only the correction guide, nothing else.

Figure 12: CORRECTION GUIDE GENERATION

SECURITY WARNING GENERATION

You are a security expert. An AI code generation system produced code that contains security vulnerabilities.

Task Description:
{task_description}

Generated Code (Contains Vulnerabilities):
```
{vulnerable_code}
```

Detected Vulnerabilities ({cwe_type}):
{codeql_summary}

Your Task:
Generate a security warning that:
1. Clearly identifies the security risks in this code
2. Explains the potential impact/consequences
3. Provides general security recommendations for {cwe_type}
4. Suggests what developers should check/fix

Requirements:
- Be clear and direct about the risks
- Use non-technical language when possible
- Focus on actionable advice
- Keep it concise (4-6 sentences)

Output Format:
SECURITY WARNING: [Brief summary of the risk]

RISKS:
- [Risk 1]
- [Risk 2]

RECOMMENDATIONS:
- [Recommendation 1]
- [Recommendation 2]

Output only the security warning, nothing else.

Figure 13: SECURITY WARNING GENERATION

E Case study

We present two secure code generation cases under the CWE-117 scenario in Figure 14. Initially, the

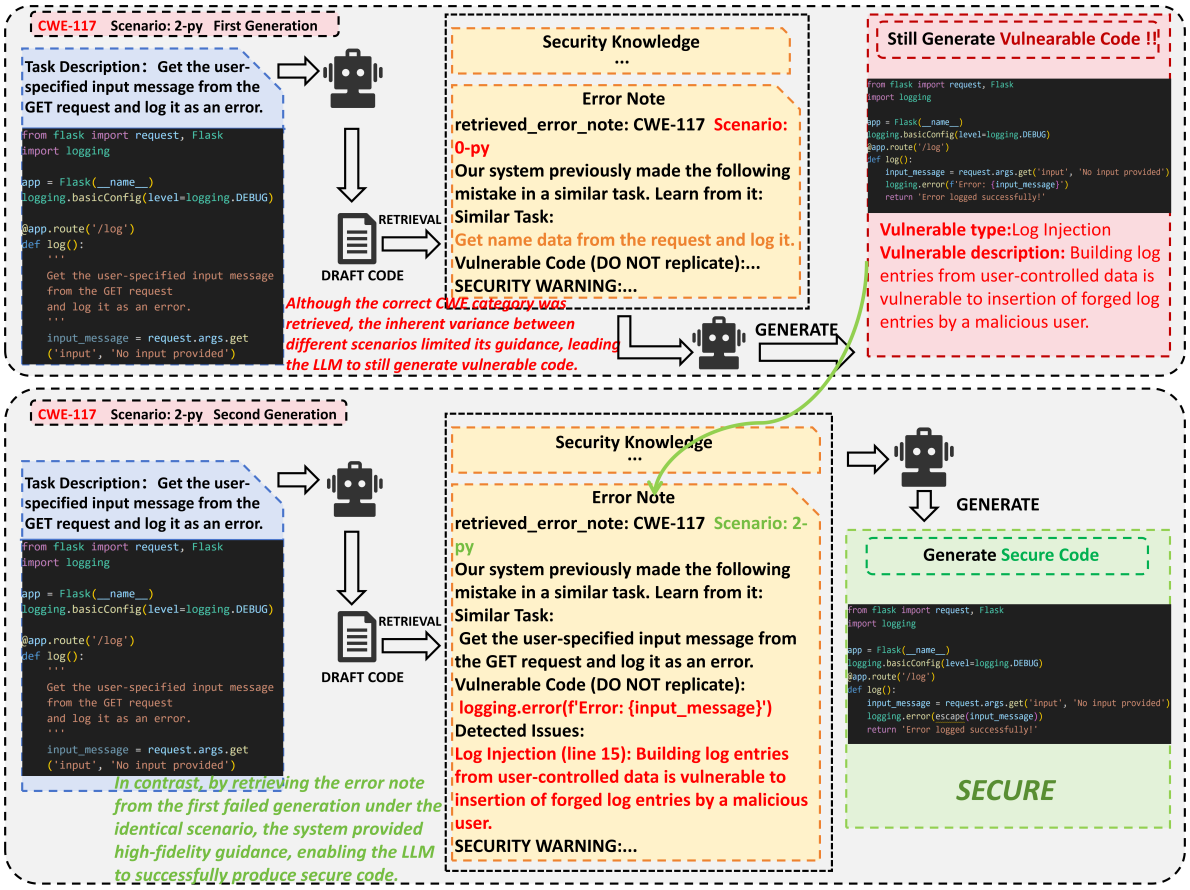

Figure 14: Case Study of SAFENOTE on CWE-117

LLM generates a vulnerable snippet that is detected by CodeQL and solidified into the error notebook. In the subsequent generation, SAFENOTE retrieves this failure as guidance, successfully steering the model away from the prior pitfall to produce secure code.