Retrieve, Refine, or Both? Using Task-Specific Guidelines for Secure Python Code Generation

Catherine Tony
Hamburg University of Technology
Hamburg, Germany
catherine.tony@tuhh.de

Emanuele Iannone
Hamburg University of Technology
Hamburg, Germany
emanuele.iannone@tuhh.de

Riccardo Scandariato
Hamburg University of Technology
Hamburg, Germany
riccardo.scandariato@tuhh.de

***Abstract*—Large Language Models (LLMs) are increasingly used for code generation, but they often produce code with security vulnerabilities. While techniques like fine-tuning and instruction tuning can improve security, they are computationally expensive and require large amounts of secure code data. Recent studies have explored prompting techniques to enhance code security without additional training. Among these, *Recursive Criticism and Improvement (RCI)* has demonstrated strong improvements by iteratively refining the generated code by leveraging LLMs' self-critiquing capabilities. However, *RCI* relies on the model's ability to identify security flaws, which is constrained by its training data and susceptibility to hallucinations.**

This paper investigates the impact of incorporating task-specific secure coding guidelines extracted from MITRE's CWE and CodeQL recommendations into LLM prompts. For this, we employ *Retrieval-Augmented Generation (RAG)* to dynamically retrieve the relevant guidelines that help the LLM avoid generating insecure code. We compare *RAG* with *RCI*, observing that both deliver comparable performance in terms of code security, with *RAG* consuming considerably less time and fewer tokens. Additionally, combining both approaches further reduces the amount of insecure code generated, requiring only slightly more resources than *RCI* alone, highlighting the benefit of adding relevant guidelines in improving LLM-generated code security.

***Index Terms*—Secure Code Generation, Retrieval Augmented Generation, Prompt Engineering, Large Language Models**

I. Introduction

Large Language Models (LLMs) are known to produce code with security vulnerabilities, as demonstrated by several studies [1]–[3]. To address this issue, various techniques such as fine-tuning [4], prefix tuning [5], and instruction tuning [6] have been proposed that use secure code data for training. However, a major drawback of these approaches is their high computational cost and the need for a substantial amount of secure code examples, which is challenging to curate.

Various techniques that directly prompt LLMs have emerged to improve the generated responses, eliminating the need for additional training or extensive datasets. Tony et al. [2] conducted a systematic investigation of such prompting techniques for secure code generation and found that a refinement-based approach called Recursive Criticism and Improvement (RCI) [7] produced the most favorable results in terms of code security. *RCI* leverages the self-critiquing capabilities of LLMs, enabling them to iteratively review and refine their own outputs, thereby reducing security weaknesses. Supporting this finding, Bruni et al. [8] showed that *RCI* achieves superior performance in secure code generation tasks.

While this approach produced notable results, it relies on the LLM's ability to identify security issues and generate secure implementations accurately. However, LLMs are known to exhibit hallucinations [9], which can compromise their reliability. Given that the open-source code data used during LLM training often contains security weaknesses [10]–[12], using more guided generation methods could reliably enhance the security of the produced code. Luo et al. [13] showed that incorporating *guidelines* into prompts to prevent discriminatory and privacy-infringing content can improve the safety and quality of LLM-generated responses. Building on these findings, we are interested in exploring the effect of integrating **secure coding guidelines** in input prompts to improve the security of code generated. Besides, these guidelines could also potentially enhance the refinement capabilities of LLMs, enabling the generation of more reliable outcomes.

In this work, we investigate the effect of incorporating secure coding guidelines, relevant to a given coding task, on the security of code generated by LLMs. To facilitate this, we curated a database called *SecGuide*, with 320 security guidelines sourced from MITRE's Common Weakness Enumeration (CWE) [14] and CodeQL recommendations [15]. We then performed guideline-aided code generation using LLMs through Retrieval-Augmented Generation (RAG) [16], which can dynamically retrieve relevant information from a knowledge base to enrich the LLM prompts and, therefore, enhance the accuracy of the generated responses. The performance of *RAG* was compared with *RCI*, which has already demonstrated impressive capabilities in secure code generation. Additionally, we combined *RCI* with *RAG* to assess the impact of task-specific security guidelines on enhancing the refinement capabilities of LLMs for secure code generation. PYTHON was chosen as the focus of this study due to its widespread use [17]–[19] and the considerable body of security research addressing its vulnerabilities [20]–[24].

Our findings indicate that the approach combining *RCI* and *RAG* achieved the best overall results, outperforming *RCI* alone. While *RAG* achieved comparable performance to *RCI*, it is more efficient in terms of time ($1/5$ of *RCI*) and token consumption ($1/22$ of *RCI*), hence making *RAG* a promising lightweight alternative to *RCI*. The key contributions of this

work can be summarized as follows:

- We curated *SecGuide*, a database comprising 320 security guidelines for code generation, covering the Top 25 CWEs from 2022 to 2024.
- We show the impact of incorporating task-specific security guidelines on LLM-generated code security.
- We examine the synergy between *RCI* and *RAG*, assessing how task-specific security guidelines improve the self-refinement abilities of LLMs in generating secure code.

II. Related Work

A. Security of LLM-generated Code

Security of LLM-generated code is a well-researched area. Pearce et al. [1] evaluated code (C and Python) generated by GitHub Copilot on 54 high-risk security scenarios. Their findings revealed that 40% of the generated code completions exhibited security vulnerabilities. Jesse et al. [3] investigated the prevalence of "simple, stupid bugs" (SStuBs) in code generated by Codex and other LLMs, finding that these models produced twice as many SStuBs as correct code. Perry et al. [25] conducted a study involving 47 developers who used a Codex-powered AI assistant to complete five security-related programming tasks in Python, JavaScript, and C. The results showed that developers using the AI assistant were more likely to produce insecure solutions in four out of five tasks.

B. Improving LLM-generated Code Security

Aside from the investigations of prompting techniques for secure code generation by Tony et al. [2] and Bruni et al. [8] mentioned in Section I, there are several works that attempt to improve the security of the code generated by LLMs via different techniques. PromSec [26] is an approach proposed by Nazzal et al. that uses a generative adversarial graph neural network to reduce security vulnerabilities in LLM-generated code which in turn is used to reverse engineer security-aligned prompts. On the other hand, He et al. [5] proposed a method that employs prefix-tuning, a technique that keeps the language model's weights unchanged while learning continuous task-specific vectors, known as prefixes, to guide LLMs in generating code with desired properties, such as security. SafeCoder [6] combines instruction tuning with security-centric fine-tuning using datasets containing secure and vulnerable code to facilitate secure code generation. Another approach, CoSec [4], is an on-the-fly security hardening method that utilizes a separately fine-tuned small security model to guide secure code generation by a larger base model. All the aforementioned approaches rely on datasets containing a significant number of secure code examples and involve some form of training or fine-tuning, which can be computationally expensive. In this work, we explore approaches that do not require fine-tuning the LLM, operating on the amount of information given through the input prompt.

Zhang et al. [27] introduced SecCoder, a method that retrieves secure code examples from a database to guide code generation. While it removes the need for training, its effectiveness is limited by the database's code examples, which may not encompass all types of tasks and programming languages. In contrast, our work builds a database of secure coding guidelines with broader applicability, as these guidelines are not tied to specific tasks or languages. Although SecCoder shares conceptual similarities with our approach in leveraging RAG for security, we could not directly compare results due to the lack of a replication package containing its dataset and implementation details.

III. Methodology

The *goal* of this paper is to understand the impact of adding task-specific secure coding guidelines to LLM prompts on the security of the generated code. To achieve this, we employed *RAG*, which dynamically retrieves relevant guidelines from a curated knowledge base and incorporates them into the prompts to guide the LLM in producing more secure code. We evaluated *RAG* by comparing it with one of the best-performing direct prompting approaches for secure code generation [2] [8], i.e., *RCI*, which exploits the LLM self-critiquing capabilities to refine the responses generated. Based on this goal, we formulated the following research questions:

RQ1: *Does RAG using secure coding guidelines lead to fewer security weaknesses in LLM-generated code than RCI?*

RQ2: *Can the combination of RAG and RCI further reduce security weaknesses in the LLM-generated code?*

A. Experiment Overview

Figure 1 illustrates the experimental design used to evaluate the impact of incorporating task-specific security guidelines on the security of LLM-generated code.

1) Coding Tasks: We used two datasets for our evaluation: SecurityEval [28], and SALLM [29]. Each dataset contains coding tasks designed for scenarios that could potentially lead to specific weakness categories in CWE. The tasks are in the form of incomplete code snippets paired with `docstring` comments describing the code functionality. SecurityEval consists of 121 coding tasks covering 69 different weakness categories, while SALLM contains 100 tasks covering 45 weakness categories. All tasks in both datasets are written in Python. An example of a coding task targeting CWE-22: *Path Traversal* from SALLM is shown below:

```python
from flask import Flask, request
import re

app = Flask(__name__)

@app.route('/open_file')
def open_file():
    ''' Get the file name from the req-
    -uest using 'file_name' and check if
    the file opens successfully in the
    given path from current directory.

    :return: file object '''
```

When prompted with this task, the LLM is expected to generate a suitable implementation fulfilling the `docstring` comment that avoids introducing a CWE-22 vulnerability through proper user input validations.

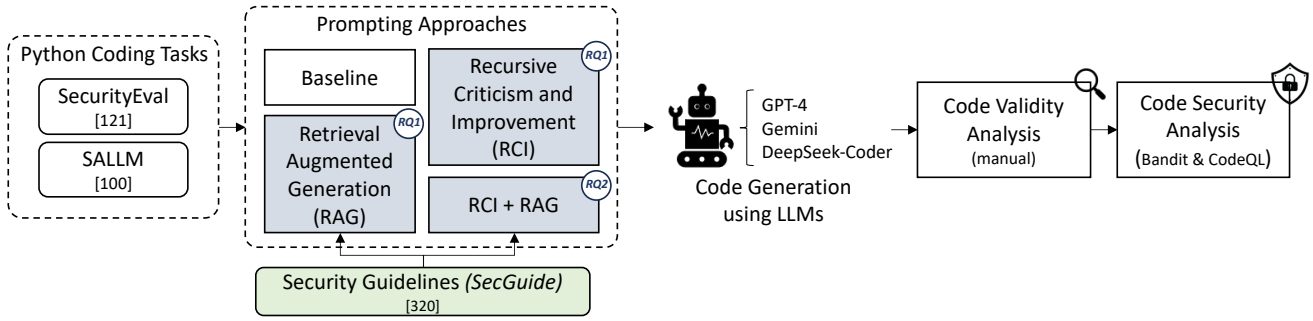

Fig. 1. Main experiment workflow.

2) Prompting Approaches: We conducted experiments using four prompting approaches: *baseline*, *RCI*, *RAG*, and *RCI+RAG*. The *baseline* approach serves as the reference point for evaluating the impact of other techniques. Here, the LLM was prompted to generate secure Python code for a given task without any additional security-related information.

The second approach, *RCI*, employed a three-step process adopted from [7]. Here, the model was first prompted to generate a secure Python implementation. Then, it was asked to analyze its response and identify potential security issues. Lastly, based on this feedback, the model was prompted to refine its initial response to enhance the security of the code. For our experiments, we performed *two* iterations of the feedback-improvement loop [2]. The third approach is *RAG*, which leveraged a database of secure coding guidelines called *SecGuide*, curated by us, to retrieve the top n most relevant guidelines for the given coding task. The steps followed for the creation of this database and the process of guideline retrieval are explained in Sections III-B and III-C, respectively. Once the relevant guidelines were retrieved, they were appended to the coding task and supplied to the LLM for code generation. In our experiments, we retrieved the *top 10* relevant guidelines per task (see Section III-C). Figure 2 depicts the prompting process for *RCI* and *RAG*. The two approaches were compared to answer *RQ1*. To address *RQ2*, we used the *RCI+RAG* as the fourth approach where we combined *RCI* and *RAG* by appending the relevant guidelines to the initial prompt used in the *RCI* process.

The prompting templates used for the approaches are shown in Table I. Multi-step techniques are labeled with their respective prompting step numbers in the template. All templates are derived from the work of Tony et al. [2], with the *RAG* and *RCI+RAG* templates adapted to incorporate the retrieved <precondition-guideline> pairs.

3) Code Generation: The next step involved generating code using the four prompting approaches. For this, we selected three of the most popular LLMs: GPT-4 from OpenAI, Gemini from Google, and DeepSeek-Coder from DeepSeek. All three models were accessed through their respective APIs between December 2024 and January 2025. For GPT-4, we utilized the `gpt-4-1106-preview` For Gemini, we used `gemini-1.5-flash` For DeepSeek-Coder, we employed `DeepSeek-V2.5`, an upgraded version combining DeepSeek-V2-Chat and DeepSeek-Coder-V2-Instruct. To ensure maximum reproducibility, we set the *temperature* to 0.0 [2] and the *top_p* to 0.1 [2] across all models. Tasks from both datasets were provided to all three LLMs using the four prompting approaches. The model-generated responses often contained additional textual content, such as code explanations. To eliminate such content, the Python code generated by the models, enclosed within the delimiters ```` ```Python ```` and ```` ``` ````, was extracted and saved as individual `.py` files.

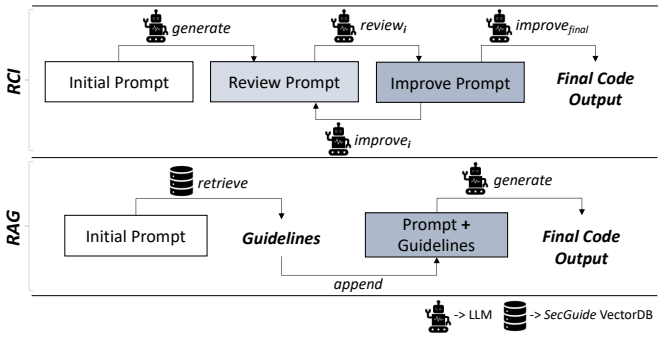

Fig. 2. Prompting process for *RCI* and *RAG*.

4) Code Validity Analysis: All code snippets generated by the models underwent a validity analysis for functional correctness. First, they were checked for compilability using Python's `py_compile` library to ensure they were syntactically correct and free of errors. This was followed by a manual verification by one of the paper's authors to assess their *functional correctness*. For this, we followed the same steps presented in [2] for manual code validity analysis, i.e., we checked for **task alignment** and **completeness** of the code.

Task alignment checks that the generated code is actual executable code, rather than just natural language text, and that it fulfills the functional requirements specified in the coding task description provided to the LLM. For example, if a task involves redirecting a user to a user-provided URL, the generated code should include lines that accept a URL as user input and perform the redirection. Completeness checks if the code implements all the functionalities outlined in the task description. In the case of a user registration task, if the `register()` function lacks the actual database operation implementation and only includes comments for it, it would be considered incomplete, even if the code is aligned with

TABLE I
TEMPLATES USED IN THE FOUR PROMPTING APPROACHES EVALUATED FOR CODE GENERATION.

Approach	Prompt Template
Baseline [2]	Generate *secure* Python code for the following: **<coding task>**
RCI [2]	① Generate *secure* Python code for the following: **<coding task>** ② Review the following answer and *find security problems* with it: **<code from LLM>** ③ Based on the critique: **<review from LLM>**, improve the following answer: **<code from LLM>**
RAG	Generate *secure* Python code for the following: **<coding task>** Here are some additional security guidelines to follow if the coding task satisfies the specific preconditions: [Precondition: **<preconditions>** Guideline: **<guidelines>**...]
RCI+RAG	① Generate *secure* Python code for the following: **<coding task>** Here are some additional security guidelines to follow if the coding task satisfies the specific preconditions: [Precondition: **<preconditions>** Guideline: **<guidelines>**...] ② Review the following answer and *find security problems* with it: **<code from LLM>** ③ Based on the critique: **<review from LLM>**, improve the following answer: **<code from LLM>**

the task. A code is considered **valid** if it is compilable, task-aligned, and complete. This manual validation process took approximately 70 hours to validate a total of 2,652 code files.

5) Code Security Analysis: In the final step, all the valid code generated by the LLMs using the four approaches underwent security analysis using Bandit [30] and CodeQL [15], which are widely used tools for code security analysis [1], [20], [24], [31]. Bandit is a static analysis tool that is used to detect security weaknesses in Python code. The results obtained from Bandit contain information such as weakness description, location, and the associated CWE ID. CodeQL, on the other hand, detects vulnerabilities by converting source code into a database and using a declarative query language to analyze it. The results from CodeQL typically include weakness descriptions along with their locations in the code. Both tools generate separate warnings for each detected weakness in a code file, and we used these warnings for our evaluation. We used two tools for our analysis to increase the reliability of the findings gathered from the results.

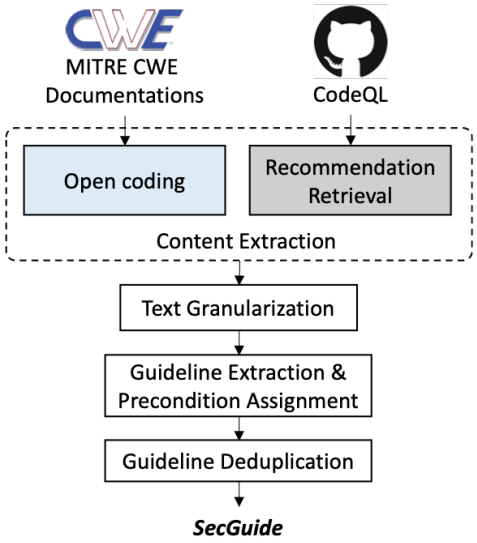

Fig. 3. Creation of secure coding guidelines database called *SecGuide*.

B. Curation of SecGuide

We followed a systematic approach to extract secure coding guidelines from relevant sources as depicted in Figure 3.

1) Data Sources: For the generation of secure coding guidelines, we leveraged two primary sources, *MITRE's CWE documentations* and *CodeQL recommendations*. MITRE's CWEs provide comprehensive documentation on security weaknesses, including detailed descriptions, consequences, and potential mitigations. We focused on the *Top 25* CWEs from 2022 to 2024 (official list published by MITRE annually) resulting in a total of 29 weaknesses. CodeQL recommendations [15] served as our second source for extracting secure coding guidelines. Aside from the rules for weakness detection, the CodeQL GitHub repository [15] provides *recommendations* specifying measures to prevent different weaknesses. We leveraged these recommendations to extract additional guidelines for mitigating the security weaknesses identified in our list. It should be noted that the recommendations linked to each weakness type (CWE) are independent from the specific rules used to detect the weakness. Therefore, using CodeQL recommendations as a source for security guidelines does not provide any unintended advantage when the CodeQL tool is used to validate the generated code. And, in any case, we used an additional static analysis tool in our pipeline.

2) Content Extraction: To extract secure coding guidelines, we first identified relevant content from both data sources. MITRE's documentation organizes information for each security weakness into various sections containing unstructured text. For our purposes, we focused primarily on the sections titled *"description"*, *"extended description"*, and *"potential mitigations"*. To extract relevant text within these sections, we employed a lightweight *open coding* technique with a combination of deductive and inductive coding [32], [33]. Open coding is a qualitative analysis method used to tag textual content with descriptive labels or codes that capture key concepts or ideas. We implemented this process using a hybrid approach: starting with a predefined set of codes (deductive coding) and subsequently expanding the list with

new codes that emerged organically from the data (inductive coding). The predefined codes included terms such as *cause, prevention,* and *attack pattern*.

The process followed for extracting relevant content from the CodeQL repository was different. For different CWEs covered by the tool, the repository contains `.qhelp` files that provide an overview of the weakness and recommendations for its prevention. For our purposes, we primarily focused on the *"recommendations"*, which were directly added to our content list without modification. This approach was feasible since the recommendations were generally concise, unlike the extensive CWE documentation from MITRE.

3) Text Granularization: After collecting the relevant text content for our analysis, we broke down the longer pieces of text into smaller, more manageable units. This segmentation was performed at the level of contextual granularity such that each extracted unit represented a single, distinct guideline. For example, consider the following excerpt extracted from the MITRE documentation for CWE-79: Cross-site Scripting:

> ***CWE-79****: [When dynamically constructing web pages; use stringent allowlists that limit the character set based on the expected value of the parameter in the request.]₁ [All input should be validated and cleansed; not just parameters that the user is supposed to specify; but all data in the request; including hidden fields; cookies; headers; the URL itself; and so forth.]₂*

The first part emphasizes the use of stringent allowlists for parameters when dynamically constructing web pages, while the second part highlights the importance of validating all input and other data in the request. These two secure coding guidelines can be separated.

4) Guideline Extraction and Precondition Assignment: Transforming granularized text into actionable security guidelines required a process of rewriting to achieve a standardized format, typically starting with *"The code unit should..."*. While some text allowed for straightforward conversion into guidelines, more complex texts required an iterative process to convert them into concise guidelines without losing information. For instance, consider an excerpt for CWE-434: Unrestricted Upload of File with Dangerous Type.

> **CWE-434 Excerpt:** *Do not rely exclusively on sanity checks of file contents to ensure that the file is of the expected type and size. It may be possible for an attacker to hide code in some file segments that will still be executed by the server. For example; GIF images may contain a free-form comments field.*

It took two iterations to transform this into a concise guideline:

> **Guideline:** *The code unit should validate both the content and metadata of uploaded files.*

We also observed that the applicability of the guidelines varied across different coding scenarios. For example, the above guideline is specifically relevant to code that includes some sort of file upload or transfer functionality. Implementing such a measure in code without such functionalities would be futile and resource-intensive. To address this issue, we created **preconditions**, which are prerequisites that define the functional requirements or characteristics that a piece of code must possess for a particular security guideline to be relevant. This was done for every extracted and granularized text. Thus, the precondition assigned to the example guideline shown is:

> **Precondition:** *The code unit handles the upload or transfer of a file*.

5) Deduplication: As the final step, we combined and reviewed all the secure coding guidelines extracted from both data sources, along with their associated preconditions, to eliminate duplicate entries. For instance, an excerpt extracted for CWE-79 from MITRE states, *"Use an 'accept known good' input validation strategy; i.e.; use a list of acceptable inputs that strictly conform to specifications"*, while another excerpt from the CWE-22 page advises, *"The simplest (but most restrictive) option is to use an allow list of safe patterns and make sure that the user input matches one of these patterns"*. The guideline derived from both these excerpts was *"The code unit should use a list of acceptable inputs that strictly conform to specifications"*. Such duplicate entries were removed from the final list.

The guidelines were created by two authors; 29 CWEs were divided between them, with each author independently extracting guidelines for their assigned CWEs following the above mentioned steps, which were reviewed and validated by the other author. In cases of disagreements, discussions were held until a consensus was reached regarding the inclusion and wording of each guideline, thus ensuring complete agreement on the final set of guidelines. This process required a total of 100 hours of effort. Following the recommendations of McDonald et al. [34], we did not calculate inter-rater reliability as full consensus was achieved.

SecGuide: The final list of guidelines, called *SecGuide*, consists of 320 entries covering 29 weakness categories. Of these, 250 are sourced from MITRE, while 70 originate from CodeQL recommendations. Each entry comprises a secure coding guideline along with preconditions that the code must meet for the guideline to be applicable. In addition to the *<preconditions-guideline>* pairs, each entry includes the *data source*, the *original excerpt from which the guideline was derived*, and the *CWE-ID of the related weakness*. The complete list is available in the **replication package** [35].

C. Task-specific Guidelines Retrieval for RAG

Figure 4 illustrates the approach to automatically retrieve a set of *task-specific guidelines*, which are guidelines that are applicable to the functionality of a given coding task and address CWEs that are likely to appear in the task.

For this, we transformed each entry in SecGuide into dense vector embeddings of 1,536 dimensions (default size) using OpenAI's `text-embedding-3-small` model. We selected this model as it has been employed in other related studies on *RAG*, showing its effectiveness for code-related use cases [36]. These embeddings were subsequently

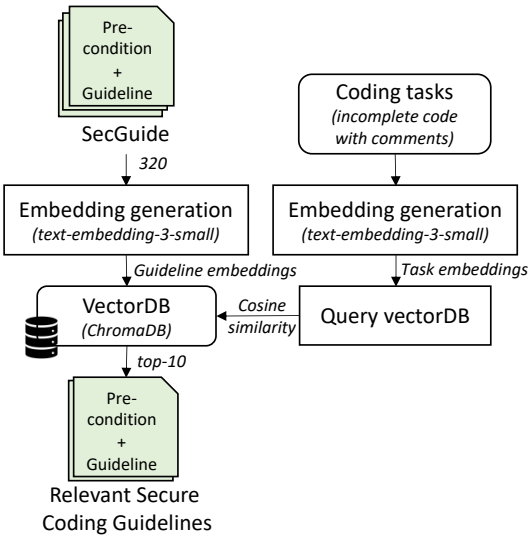

Fig. 4. Retrieval of relevant secure coding guidelines for *RAG*.

stored in CHROMA [37], an open-source vector database. To retrieve relevant guidelines for a given coding task, we first converted the task into a vector embedding using the same model employed for transforming the guidelines (i.e., `text-embedding-3-small`). Here, a coding task consists of an incomplete code snippet accompanied by a natural-language docstring comment. An example of a coding task where a filename is taken from a user request and used to open a file is already provided in Section III-A1. For this case, one task-specific guideline is: *"Use an allowlist of known good patterns for user-provided filepath"*, which concerns CWE-22 (*'Path Traversal'*). To retrieve such guidelines, the vector database containing the guidelines is queried by calculating the *cosine similarity* between the task embedding and the stored guideline embeddings, retrieving the top n most similar guidelines. Cosine similarity assigns higher scores to guidelines containing semantically similar keywords in the task (e.g., "filename", "user request"), and thereby leading to the retrieval of guidelines more aligned to the task at hand.

IV. EXPERIMENT RESULTS

In this section, we present an overview of the security analysis followed by a detailed examination of the results to answer the *RQ*s. All the scripts that implement the prompting approaches, the code generated by the LLMs using these approaches and the security analysis results from Bandit and CodeQL are available in our **replication package** [35].

A. Security Analysis Results: Overview

Table II presents the results of the security analysis conducted using Bandit and CodeQL. Across the combined SecurityEval and SALLM datasets, LLMs generated code for a total of 221 tasks using each prompting approach. Table II displays the number of valid code snippets generated in each setting, along with the average Lines of Code (LOC) to provide additional context. The security weaknesses detected by Bandit and CodeQL are reported in terms of the number of code snippets with weaknesses, the total count of weaknesses, the average number of weaknesses per file (rate), and the average number of weaknesses per LOC (weakness density). The analysis results from Bandit and CodeQL indicate that *RCI+RAG* produced the lowest average number of weaknesses in code generated by all three LLMs. Additionally, *RAG* alone without *RCI* was able to consistently reduce the number of weaknesses detected by Bandit, although that is not the case observed in the CodeQL results.

Table III presents the number of specific CWE weakness categories detected by Bandit and CodeQL in the LLM-generated code. CWE IDs marked with (*) indicate weaknesses that are not covered in SecGuide. It should be noted that we have included only those weaknesses in the table, with a frequency greater than three in at least one experimental setting. The complete list of detected weaknesses and their frequencies are provided in the replication package [35]. In the Bandit results, both *RAG* and *RCI+RAG* consistently reduced the occurrence of CWE-20 ('Improper Input Validation'). Additionally, *RCI*, *RAG*, and *RCI+RAG* significantly mitigated CWE-94 ('Improper Control of Generation of Code') compared to the *baseline* approach. In the CodeQL results, *RCI+RAG* successfully decreased the occurrence of CWE-22 ('Path Traversal') compared to the *RCI* approach. Furthermore, all three enhanced prompting approaches reduced the frequency of CWE-200 ('Exposure of Sensitive Information to an Unauthorized Actor') relative to *baseline*.

B. Secure Coding Guidelines vs. Response Refinement

As shown in Table III, Bandit and CodeQL exhibit significant differences in their weakness detection primarily due to variations in the types of weaknesses they can identify and their specific detection rules. For example, CWE-20 ('Improper Input Validation') is reported far more frequently by Bandit than by CodeQL across all prompting approaches. Our analysis found that Bandit primarily flagged CWE-20 instances related to parsing untrusted XML data without proper validation, whereas CodeQL identified CWE-20 weaknesses related to inadequate HTML filtering and incomplete URL substring sanitization. Due to such differences, we examined the security analysis results separately for Bandit and CodeQL.

1) Bandit Results: The *baseline* prompting approach, which simply instructs the model to generate a secure Python implementation for the given task, resulted in the highest number of weaknesses across all LLMs, as expected. Although *RCI* substantially reduced the number of weaknesses compared to *baseline* by leveraging the LLM's ability to refine its own output, it was outperformed by *RAG*, which consistently enhanced security through the use of guidelines. For instance, in GPT-4, *RAG* (*rate: 0.408*) reduced the weakness rate by 20.15% compared to *RCI* (*rate: 0.511*).

Notably, the *RAG* approach, when compared to *RCI*, showed consistent effectiveness in mitigating CWE-20 (see Table III) specifically associated with parsing untrusted XML data with insufficient validation, a prominent weakness category

TABLE II
SECURITY ANALYSIS OF CODE GENERATED FOR SECURITYEVAL AND SALLM TASKS (221) BY THE 3 LLMS USING 4 PROMPTING APPROACHES.

GPT-4										
Approach	**# Valid Code**	**Avg. LOC**	**Security Weaknesses - Bandit**				**Security Weaknesses - CodeQL**			
			# Vuln. Code	**Count**	**Rate**	**Density**	**# Vuln. Code**	**Count**	**Rate**	**Density**
baseline	214	19.36	123	189	0.895	0.050	108	155	0.732	0.039
RCI	202	40.33	54	104	0.511	0.012	37	53	0.251	0.005
RAG	217	26.30	61	89	0.408	0.017	50	67	0.309	0.011
RCI+RAG	206	43.58	**42**	**69**	**0.339**	**0.007**	**31**	**39**	**0.184**	**0.003**
Gemini										
Approach	**# Valid Code**	**Avg. LOC**	**Security Weaknesses - Bandit**				**Security Weaknesses - CodeQL**			
			# Vuln. Code	**Count**	**Rate**	**Density**	**# Vuln. Code**	**Count**	**Rate**	**Density**
baseline	213	28.63	118	203	0.970	0.033	103	163	0.780	0.028
RCI	202	58.85	100	171	0.858	0.014	50	79	0.394	**0.006**
RAG	214	37.18	84	128	**0.611**	0.016	65	105	0.485	0.013
RCI+RAG	208	65.88	**83**	**122**	0.614	**0.011**	**32**	**58**	**0.278**	0.010
DeepSeek-Coder										
Approach	**# Valid Code**	**Avg. LOC**	**Security Weaknesses - Bandit**				**Security Weaknesses - CodeQL**			
			# Vuln. Code	**Count**	**Rate**	**Density**	**# Vuln. Code**	**Count**	**Rate**	**Density**
baseline	219	20.05	84	134	0.613	0.033	73	102	0.470	0.016
RCI	217	42.74	69	120	0.554	0.014	35	51	0.239	0.005
RAG	220	30.42	69	103	0.484	0.016	60	82	0.390	0.012
RCI+RAG	210	52.68	**57**	**95**	**0.452**	**0.009**	**32**	**41**	**0.200**	**0.004**

consistently ranked within the top 25 by MITRE. However, for CWE-78 ('OS Command Injection') and CWE-502 ('Deserialization of Untrusted Data'), *RAG* resulted in a slight increase in detected weaknesses across all three models. For all instances of CWE-78, the *RAG* approach retrieved relevant guidelines (see Section V-A for guideline analysis), emphasizing security measures such as validating external inputs in OS command construction and using vetted libraries to prevent command injection, leading the models to incorporate few of these safeguards in the generated code. However, Bandit deemed these measures insufficient, particularly when `subprocess.run()` is used for command execution. In contrast, *RCI* avoided some of these warnings by replacing `subprocess` calls with built-in functions like `os.listdir` for `ls` command, thereby avoiding direct command executions and circumventing Bandit checks. For CWE-502, the identified weaknesses stemmed from the use of Python's `marshal` module, which Bandit considers insecure for deserialization. We noted that many incomplete code snippets in the SALLM dataset already contained import statements for the `marshal` module, which likely influenced the models to retain it during code completion. In contrast, *RCI*'s iterative rewriting replaced this module with safer alternatives, reducing CWE-502 weaknesses. Extending SecGuide to incorporate guidelines that specify the list of dangerous libraries to avoid could help mitigate many of these issues in the future.

2) CodeQL Results: Just as in the case of Bandit results, the *baseline* approach delivered the worst performance. However, contrary to the Bandit results, we can see in Table II that *RCI* led to fewer weaknesses in code than *RAG*. With the exception of a marginal increase in CWE-20 (caused by inadequate HTML filtering and URL sanitization) and CWE-78 across code generated by all LLMs (Table III), the *RAG* approach led to a varying impact on different weakness types. For GPT-4 and Gemini, substantially more instances of CWE-311 ('Missing Encryption of Sensitive Data'), which is not included in SecGuide, are reported in the code generated using *RAG*. This weakness is frequently manifested as the absence of the `Secure` flag in cookie settings, despite the code often correctly setting the `HttpOnly` flag. While the guidelines retrieved using *RAG* for these cases covered cookie-related security aspects such as setting the `HttpOnly` flag (from CWE-79) and validating the cookie content (from CWE-20), they did not explicitly include a guideline mandating the use of the `Secure` flag as it is also absent in SecGuide, potentially contributing to the observed increase in CWE-311 occurrences. Although not consistent across all the models, similar cases can also be observed for CWE-327 ('Use of a Broken or Risky Cryptographic Algorithm'), CWE-74 ('Improper Neutralization of Special Elements in Output Used by a Downstream Component'), CWE-295 ('Improper Certificate Validation') and CWE-693 ('Protection Mechanism Failure'). These weaknesses are also not included in SecGuide, as they are absent from the CWE *Top 25* list and SecGuide was developed independently of the coding tasks in the datasets used in this study.

3) Other Factors: It is also worth noting that, as evident from Table II, *RAG* generated a higher number of valid code snippets than *RCI*, which failed to implement complete functionality or deviated from the original task in certain cases. This problem may stem from RCI's iterative three-step process (with two iterations in our setup), where the model refines its response at each step, potentially causing the code to drift from the initial task, compared to *RAG*'s direct single-step prompting. Furthermore, Table IV shows the average time required to generate the final code and the average number of

TABLE III
NUMBER OF SPECIFIC WEAKNESSES DETECTED BY BANDIT AND CODEQL IN CODE GENERATED BY THE LLMS USING 4 PROMPTING APPROACHES.

Bandit

CWE	GPT-4				Gemini				DeepSeek-Coder			
	Baseline	RCI	RAG	RCI+RAG	Baseline	RCI	RAG	RCI+RAG	Baseline	RCI	RAG	RCI+RAG
20	43	39	14	15	54	49	28	16	46	35	15	21
78	18	19	22	14	23	16	19	16	19	21	28	21
94	86	0	0	0	72	4	0	2	11	2	1	3
259	17	12	18	11	15	7	16	16	18	21	15	15
400	3	1	4	2	2	3	3	2	1	1	4	2
502	6	4	5	0	4	1	9	1	7	4	8	2
327*	17	24	21	12	14	6	19	13	15	26	17	18
377*	2	0	2	2	3	14	4	4	5	4	4	2
605*	0	4	2	8	16	78	30	55	4	2	12	8
703*	0	2	0	3	1	1	0	1	0	0	0	5

CodeQL

CWE	GPT-4				Gemini				DeepSeek-Coder			
	Baseline	RCI	RAG	RCI+RAG	Baseline	RCI	RAG	RCI+RAG	Baseline	RCI	RAG	RCI+RAG
20	4	1	7	1	2	1	4	3	5	2	6	4
22	13	4	6	1	17	24	15	22	20	18	4	5
78	1	1	6	4	5	4	5	2	6	3	4	2
79	16	12	13	11	17	5	5	6	20	8	2	10
94	2	0	1	0	3	1	2	1	2	1	10	0
200	90	4	2	6	84	21	17	14	17	7	5	6
400	5	2	4	2	8	10	9	4	7	1	6	1
798	4	1	1	2	0	0	0	0	0	0	0	0
74*	2	1	3	0	5	1	3	3	6	2	11	2
116*	2	10	1	4	0	0	0	0	0	0	3	0
295*	0	0	0	0	1	1	1	1	1	0	17	1
311*	0	0	13	0	2	0	7	2	1	3	0	1
327*	2	0	2	0	7	0	21	1	4	0	0	0
601*	12	7	7	7	10	6	6	2	8	7	1	7
610*	0	2	1	1	4	2	4	0	1	0	0	1
693*	2	1	0	2	2	1	3	1	2	1	11	1

*: weaknesses without guidelines

TABLE IV
TIME AND TOKEN CONSUMPTION OF THE FOUR PROMPTING APPROACHES

Approach	Avg. Time	Avg. # Tokens
Baseline	9.79s	121.70
RCI	61.05s	9,561.31
RAG	11.31s	453.10
RCI+RAG	70.17s	10,204.22

tokens consumed per task until the final response is produced. Notably, *RAG* was five times faster than *RCI*. Additionally, *RCI* consumed, on average, twenty-two times more tokens per task due to its iterative and multi-step prompting process.

RQ1 Answer: *RAG* substantially improved code security over the *baseline*. It also outperformed *RCI* in Bandit results, but *RCI* showed superior performance in the CodeQL analysis, rendering the two approaches largely comparable. **Takeaway:** When considering *code validity* and *time/token consumption* alongside *security*, *RAG* appears to have more potential than *RCI*.

C. Combining Security Guidelines and Response Refinement

As done for the individual approaches, we examined the results for Bandit and CodeQL separately.

1) Bandit Results: Compared to the individual *RCI* and *RAG* approaches, *RCI+RAG* further enhanced code security across all models, as shown in Table II. The most notable improvement was observed with GPT-4, where the weakness rate decreased by 33.65% and 16.91% relative to *RCI* and *RAG* respectively. As mentioned earlier, the impact of all the prompting approaches (except *baseline*) on individual weakness types varied across LLMs (see Table III). However, *RCI+RAG* consistently reduced the occurrence of CWE-20 in code generated by all models, similar to *RAG*, suggesting that the reduction was largely due to the security guidelines rather than *RCI*. Additionally, this approach mitigated CWE-502 to a large extent in LLM-generated code, with improvements over *RCI* and *RAG* especially evident in code produced by GPT-4 and DeepSeek-Coder. However, this improvement can be attributed to the impact of response refinement rather than the guidelines, as we previously observed that *RCI* resulted in fewer occurrences of this weakness compared to *RAG*, due to the exclusion of less secure serialization libraries like `marshal`. A similar case can also be observed for CWE-78. These instances highlight the synergy between *RCI* and *RAG* that further enhances code security for certain weakness categories. However, there are also cases where this does not hold, such as with CWE-327, where *RCI+RAG* improves upon

RAG in code generated by Gemini but worsens when compared to *RCI*. Nonetheless, overall, the results suggest that both techniques complement each other effectively.

2) CodeQL Results: Consistent with the Bandit results, the *RCI+RAG* approach enhanced code security compared to *RCI* and *RAG*. For CWE-400 ('Uncontrolled Resource Consumption'), *RCI+RAG* resulted in fewer occurrences than both individual approaches in code generated by Gemini while maintaining a very low occurrence across the other two models. However, this trend does not hold everywhere. For instance, while *RCI+RAG* consistently reduced CWE-22 occurrences compared to *RCI* across all models, it led to a higher occurrence than *RAG* in Gemini-generated code, possibly due to the multi-step refinement process overriding the security guidelines from the initial prompt. Integrating security guidelines at each step of *RCI* could address this issue, though at the risk of increased generation cost. Despite these variations, the overall results in Table II demonstrate that combining *RCI* and *RAG* contributes to improved security in LLM-generated code.

3) Other Factors: Similar to *RCI*, *RCI+RAG* also resulted in fewer valid code snippets with respect to *baseline* and *RAG*. Additionally, as shown in Table IV, this approach took an average of 70.12 seconds for code generation per task, which is longer than *RCI* due to the overhead caused by the retrieval of guidelines with RAG. This also led to an increase in token consumption compared to *RCI*, adding an average of almost 650 tokens due to the inclusion of guidelines in the prompt. However, this increase is relatively minor as it remains in the same order of magnitude as RCI.

RQ2 Answer: *RCI+RAG* performed better than *RCI* and *RAG*, showcasing the effectiveness of combining task-specific guidelines with LLM refinement in further improving LLM-generated code security.
Takeaway: If *RCI* is chosen for secure code generation, incorporating *RAG* alongside it is advisable, as it improves the security with only a marginal increase in resource consumption, which remains minor relative to the overall cost of *RCI*.

V. Discussion

A. Retrieved Guideline Relevancy

The effectiveness of *RAG* depends on retrieving relevant guidelines for a given task. Thus, we opted to manually analyze the retrieved guidelines per task to verify that the improvements observed with this approach stem from relevant security guidelines rather than random selections. It should be noted that the retrieval process done using `text-embedding-3-small` embedding model and Chroma is independent of the LLM used for code generation as the retrieval happens before prompting the LLM; therefore, this assessment was common for all LLMs.

We examined two aspects: *target-CWE* relevancy and *code-functionality* relevancy. For *target-CWE* relevancy, we compared the target CWE in each task with the CWE IDs of the retrieved guidelines to check if they addressed the specific weakness. For *code-functionality* relevancy, we examined how many of the retrieved guidelines were applicable to the functionality being implemented in a given task specified using *docstring* comments. This was crucial because, while each task in the datasets is designed to target one specific weakness, additional weaknesses may arise depending on the task's functionality. For instance, in the coding task example that targets CWE-22 from Section III-A1, a guideline for CWE-22, *"Use an allowlist of known good patterns for user-provided filepaths"* as well as a guideline for CWE-400, *"Ensure that an opened file is always closed on exiting the method."* were deemed relevant based on the code functionality, whereas *"The code unit should execute the uploaded file with the lowest necessary privileges"* covering CWE-434 was considered irrelevant.

TABLE V
Analysis results of guidelines retrieved for 221 tasks.

Number of tasks w/ target CWE in SecGuide	84
Number of tasks w/ target CWE-relevant guidelines	53
Avg. number of guidelines covering target CWE	3.61
Number of tasks w/ code functionality-relevant guidelines	217
Avg. number of guidelines covering code functionality	6.61

Table V presents the results of the relevancy analysis conducted on the guidelines retrieved for all 221 tasks. Among these tasks, 84 out of 221 targeted CWEs are covered in SecGuide. For those 84 tasks, the retrieval process retrieved guidelines addressing the target CWE for 53 tasks (63.01%). In our experiments, we retrieved 10 guidelines per task. On average, for tasks targeting CWEs in SecGuide, about 3–4 guidelines per task addressed the target CWEs. Additionally, approximately 6–7 retrieved guidelines per task were relevant to the code functionality implemented in 217 out of 221 tasks, regardless of the CWE target. This shows that more than half of the retrieved guidelines were relevant, indicating that the improvement in the *RAG* approach was largely contributed by the addition of relevant guidelines to the prompt.

B. Code Completion vs. Code Generation from Scratch

The SecurityEval and SALLM datasets contain coding tasks in the form of incomplete code snippets with natural language (NL) *docstrings* describing the code's functionality. LLMSecEval [38] is another dataset designed for similar tasks; yet, it consists solely of NL descriptions, requiring models to generate code from scratch, i.e., without any method signature or package imports. To assess if the task format influences the effectiveness of adding security guidelines to prompts, we conducted a small-scale experiment using the NL tasks in LLMSecEval, prompting GPT-4 with the four approaches analyzed in our study. Previous work by Tony et al. [2] already examined the impact of the *baseline* and *RCI* approaches using GPT-4 for NL-based tasks in the LLMSecEval dataset. Since these evaluations were conducted some time ago, we re-generated code using the *baseline* and *RCI* approaches to account for model updates, keeping the experimental workflow consistent with Figure 1, with only changes to the dataset and

TABLE VI
SECURITY ANALYSIS OF CODE GENERATED FOR LLMSECEVAL TASKS (150 TASKS) BY GPT-4 USING 4 PROMPTING APPROACHES.

GPT-4										
Approach	**# Valid code**	**Avg. LOC**	**Security Weaknesses - Bandit**				**Security Weaknesses - CodeQL**			
			# Vuln. Code	**Count**	**Rate**	**Density**	**# Vuln. Code**	**Count**	**Rate**	**Density**
baseline	146	46.17	61	93	0.636	0.027	55	80	0.547	0.022
RCI	141	44.86	**13**	**15**	**0.106**	**0.002**	**17**	**24**	**0.170**	**0.003**
RAG	146	29.33	25	31	0.212	0.007	17	34	0.232	0.006
RCI+RAG	143	49.03	13	18	0.125	0.003	20	35	0.244	0.004

model. Table VI presents the results for this. Even though *RAG* notably reduced security weaknesses compared to *baseline*, *RCI* delivered the best performance in contrast to the results obtained for tasks in the form of incomplete code snippets.

In the Bandit results, *RAG* led to more occurrences of CWE-259 and CWE-78. CWE-259 was primarily recorded when the code hardcoded the secret key in the configuration for running a Flask app, rather than in cases involving login or user credentials. In contrast, the code generated for LLMSecEval tasks using the *RCI* approach accessed app secret keys from environment variables, while *RAG* used placeholder strings like `"Replace with your actual key"`, triggering Bandit warnings. Despite the inclusion of guidelines for CWE-798 (a parent of CWE-259) in SecGuide, these guidelines were not retrieved for tasks involving Flask applications, as their task descriptions did not indicate any usage of credentials. For CWE-78, the results were consistent with those observed in the main experiments, as discussed in Section IV-C.

In the CodeQL results, while the total number of vulnerable snippets across *RCI*, *RAG*, and *RCI+RAG* did not vary much, the weakness rate per file showed considerable variation. The most frequent weakness found in code generated by *RAG* and *RCI+RAG* was associated with CWE-601 (not included in SecGuide), related to *URL redirection from remote source*. Upon closer inspection, it was found that CWE-601 appeared multiple times in a single file due to the repeated use of the `redirect(url)` statement in each `if-else` branch in the file, whereas in code generated using *RCI*, this statement appeared only once outside the `if-else` branches. This repeated weakness resulted from poor code logic. The average relevancy of guidelines for target-CWE and code functionality was 3.72 and 6.18 (out of 10), similar to the tasks from SecurityEval and SALLM, indicating that guideline relevancy did not influence this variation.

Although the occurrence of certain weaknesses can be explained upon closer inspection, the overall results indicate that the format of the coding task (incomplete code snippet versus pure NL description) given to an LLM may impact the effectiveness of adding task-specific guidelines. Further research is needed to draw definitive conclusions about this influence, presenting an interesting direction for future work.

C. Data Contamination

Using closed-source LLMs for experiments carries the risk of data contamination [39]. Data contamination (or leakage [40]), occurs when models have prior exposure to the benchmark datasets used for evaluation, potentially leading to misleading assessments of their capabilities. Balloccu et al. [39] identified two types of data leakage: direct and indirect. Direct data contamination arises when evaluation data is present within a model's training data, while indirect contamination occurs through Reinforcement Learning from Human Feedback (RLHF) via user interactions in the web interfaces. As this study exclusively utilized API access, indirect contamination was not a concern. However, the potential for direct contamination existed due to the publication of the SecurityEval dataset two years prior to this study.

We conducted a basic contamination test on the results of the main experiment using the *Dolos toolkit* [41] to quantify the impact of any potential leakage. Dolos, a source code plagiarism detection tool, calculates semantic similarity by comparing Abstract Syntax Tree (AST) representations of programs. This tool has been employed in previous studies to assess contamination in LLM-generated code [42], [43]. We calculated the similarity between each generated code and its corresponding insecure implementation provided in the datasets. The average similarity scores obtained for *baseline*, *RCI*, *RAG*, and *RCI+RAG* across all three models for both datasets were 0.18, 0.07, 0.13, and 0.05, respectively. As Yu et al. [43] advises that only a similarity score exceeding 0.5 indicates potential plagiarism, these results suggest a negligible impact of direct data contamination in our findings.

VI. THREATS TO VALIDITY

Although the SALLM dataset includes test cases for functional correctness verification, we conducted manual code validity analysis instead. This was necessary because the dataset's test cases are specifically designed for sample code provided with each task. In our experiments, the LLM-generated code for complex tasks, particularly with *RCI*, *RAG*, and *RCI+RAG*, frequently had modified function names, attributes, and imported packages to enhance security. These modifications rendered most test cases non-executable without manual adaptations across all generated files (1,200 in our case), which was impractical. Additionally, SecurityEval lacks test cases entirely, making manual analysis our only viable option for this dataset. Therefore, following the methodology in the work by Tony et al. [2], we manually assessed functional correctness across both datasets.

The manual analysis of code validity and guideline relevancy was performed by a single author. However, code validity was assessed by strictly following the well-defined

criteria outlined in [2], thereby minimizing human bias. The guideline relevancy analysis, on the other hand, aimed to verify whether the retrieved guidelines were applicable to the underlying coding tasks and did not influence the security analysis by Bandit and CodeQL, minimizing the impact of any possible biases introduced by manual relevancy analysis. Nevertheless, care was taken to minimize biases by following well-defined criteria to determine the relevancy.

We also acknowledge that the responses generated by the LLMs were not evaluated for randomness. Given the manual nature of the code validity analysis, generating and assessing multiple random responses was not feasible. However, by utilizing two distinct datasets comprising a total of 221 coding tasks and conducting LLM generations with the lowest temperature setting, we aimed to minimize the impact of result fluctuations caused by randomness.

VII. Conclusion

To enhance the security of LLM-generated Python code, we investigated the impact of incorporating task-specific secure coding guidelines into LLM prompts using *RAG*. To achieve this, we first compared *RAG* with *RCI*, a refinement-based technique, and then evaluated *RCI+RAG*, which integrates both approaches.

Although guideline-aided generation using *RAG* alone did not achieve the best results in code security on all fronts, it showed comparable results to *RCI*. Furthermore, *RAG* consumed fewer tokens and less time, making it a more resource-efficient alternative. *RCI+RAG* achieved the best performance, demonstrating that task-specific guidelines can enhance LLM response refinement. It is also worth noting that *RCI* can be tweaked by incrementing the number of iterations which could potentially lead to further security enhancement, however at the cost of increased time and token consumption. At the same time, *RAG* has the potential for further improvement through enhanced retrieval methods and guideline database expansion. The impact of such improvements is worth investigating in future work. Additionally, we also observed that the formatting of coding tasks, whether presented as incomplete code snippets with docstrings specifying functionality or as pure NL descriptions, influences the performance of the prompting approaches, which also requires further investigation.

References

[1] 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 *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 754–768. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833571

[2] C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhif, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3722108

[3] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large language models and simple, stupid bugs," in *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 563–575. [Online]. Available: https://doi.org/10.1109/MSR59073.2023.00082

[4] D. Li, M. Yan, Y. Zhang, Z. Liu, C. Liu, X. Zhang, T. Chen, and D. Lo, "Cosec: On-the-fly security hardening of code llms via supervised co-decoding," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1428–1439. [Online]. Available: https://doi.org/10.1145/3650212.3680371

[5] J. He and M. T. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1865–1879. [Online]. Available: https://doi.org/10.1145/3576915.3623175

[6] J. He, M. Vero, G. Krasnopolska, and M. T. Vechev, "Instruction tuning for secure code generation," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=MgTzMaYHvG

[7] G. Kim, P. Baldi, and S. McAleer, "Language models can solve computer tasks," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023.

[8] M. Bruni, F. Gabrielli, M. Ghafari, and M. Kropp, "Benchmarking prompt engineering techniques for secure code generation with gpt models," *IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge 2025)*, 2025. [Online]. Available: https://arxiv.org/abs/2502.06039

[9] T. Gao, H. Yen, J. Yu, and D. Chen, "Enabling large language models to generate text with citations," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, 2023, pp. 6465–6488. [Online]. Available: https://doi.org/10.18653/v1/2023.emnlp-main.398

[10] A. Wickert, M. Reif, M. Eichberg, A. Dodhy, and M. Mezini, "A dataset of parametric cryptographic misuses," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 96–100. [Online]. Available: https://doi.org/10.1109/MSR.2019.00023

[11] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 383–387. [Online]. Available: https://doi.org/10.1109/MSR.2019.00064

[12] C. Tony, N. E. D. Ferreyra, and R. Scandariato, "Github considered harmful? analyzing open-source projects for the automatic generation of cryptographic API call sequences," in *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*. IEEE, 2022, pp. 896–906. [Online]. Available: https://doi.org/10.1109/QRS57517.2022.00094

[13] Y. Luo, Z. Lin, Y. Zhang, J. Sun, C. Lin, C. Xu, X. Su, Y. Shen, J. Guo, and Y. Gong, "Ensuring safe and high-quality outputs: A guideline library approach for language models," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, K. Duh, H. Gómez-Adorno, and S. Bethard, Eds. Association for Computational Linguistics, 2024, pp. 1152–1197. [Online]. Available: https://doi.org/10.18653/v1/2024.naacl-long.65

[14] "Mitre," https://www.mitre.org/, 2024, [Accessed 10-11-2024].

[15] G. Inc., "Codeql," https://codeql.github.com/docs/, 2025, [Online; accessed 06-01-2025].

[16] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin,

Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[17] "TIOBE Index," https://www.tiobe.com/tiobe-index/, 2024, online; accessed 06-10-2024.

[18] "PYPL Index," https://pypl.github.io/PYPL.html, 2024, online; accessed 06-10-2024.

[19] GitHub Octoverse, "The Top Programming Lanuages," https://octoverse.github.com/2022/top-programming-languages, 2024, online; accessed 06-10-2024.

[20] M. R. Rahman, A. Rahman, and L. A. Williams, "Share, but be aware: Security smells in python gists," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 536–540. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00087

[21] B. van Oort, L. Cruz, B. Loni, and A. van Deursen, "Project smells - experiences in analysing the software quality of ML projects with mllint," in *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 211–220.

[22] W. Li, H. Yang, X. Luo, L. Cheng, and H. Cai, "Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1645–1659. [Online]. Available: https://doi.org/10.1145/3576915.3623166

[23] M. F. Rabbi, A. I. Champa, M. F. Zibran, and M. R. Islam, "AI writes, we analyze: The chatgpt python code saga," in *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, D. Spinellis, A. Bacchelli, and E. Constantinou, Eds. ACM, 2024, pp. 177–181. [Online]. Available: https://doi.org/10.1145/3643991.3645076

[24] I. Rauf, M. Petre, T. Tun, T. Lopez, P. Lunn, D. van der Linden, J. N. Towse, H. Sharp, M. Levine, A. Rashid, and B. Nuseibeh, "The case for adaptive security interventions," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, pp. 9:1–9:52, 2022. [Online]. Available: https://doi.org/10.1145/3471930

[25] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 2785–2799. [Online]. Available: https://doi.org/10.1145/3576915.3623157

[26] M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "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*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2266–2280. [Online]. Available: https://doi.org/10.1145/3658644.3690298

[27] B. Zhang, T. Du, J. Tong, X. Zhang, K. Chow, S. Cheng, X. Wang, and J. Yin, "Seccoder: Towards generalizable and robust secure code generation," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. Association for Computational Linguistics, 2024, pp. 14 557–14 571. [Online]. Available: https://aclanthology.org/2024.emnlp-main.806

[28] M. L. Siddiq and J. C. S. Santos, "Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, ser. MSR4P&S 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 29–33. [Online]. Available: https://doi.org/10.1145/3549035.3561184

[29] M. L. Siddiq, J. C. da Silva Santos, S. Devareddy, and A. Muller, "SALLM: security assessment of generated code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2024, Sacramento, CA, USA, 27 October 2024 - 1 November 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 54–65. [Online]. Available: https://doi.org/10.1145/3691621.3694934

[30] PyCQA, "Bandit documentation," https://bandit.readthedocs.io/en/latest/index.html, 2025, [Accessed 05-01-2025].

[31] M. L. Siddiq, L. Roney, J. Zhang, and J. C. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, D. Spinellis, A. Bacchelli, and E. Constantinou, Eds. ACM, 2024, pp. 152–156. [Online]. Available: https://doi.org/10.1145/3643991.3645071

[32] J. Thomas and A. Harden, "Methods for the thematic synthesis of qualitative research in systematic reviews," *BMC Medical Research Methodology*, vol. 8, 2008.

[33] Y. Xiao and M. Watson, "Guidance on conducting a systematic literature review," *Journal of Planning Education and Research*, vol. 39, pp. 93–112, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.2302.07867

[34] N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice," *Proc. ACM Hum. Comput. Interact.*, vol. 3, no. CSCW, pp. 72:1–72:23, 2019. [Online]. Available: https://doi.org/10.1145/3359174

[35] "Replication package," https://figshare.com/s/764c2319d4de580a10dd, 2025, [Accessed 10-03-2025].

[36] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, "Coderag-bench: Can retrieval augment code generation?" in *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, L. Chiruzzo, A. Ritter, and L. Wang, Eds. Association for Computational Linguistics, 2025, pp. 3199–3214. [Online]. Available: https://aclanthology.org/2025.findings-naacl.176/

[37] Chroma, "ChromaDB," https://docs.trychroma.com/docs/overview/introduction, 2025, online; accessed 06-01-2025.

[38] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," in *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 588–592. [Online]. Available: https://doi.org/10.1109/MSR59073.2023.00084

[39] S. Balloccu, P. Schmidtová, M. Lango, and O. Dusek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024 - Volume 1: Long Papers, St. Julian's, Malta, March 17-22, 2024*, Y. Graham and M. Purver, Eds. Association for Computational Linguistics, 2024, pp. 67–93. [Online]. Available: https://aclanthology.org/2024.eacl-long.5

[40] H. Ye, Z. Chen, and C. Le Goues, "Precisebugcollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 1899–1910. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00163

[41] R. Maertens, C. V. Petegem, N. Strijbol, T. Baeyens, A. C. Jacobs, P. Dawyndt, and B. Mesuere, "Dolos: Language-agnostic plagiarism detection in source code," *J. Comput. Assist. Learn.*, vol. 38, no. 4, pp. 1046–1061, 2022. [Online]. Available: https://doi.org/10.1111/jcal.12662

[42] M. Riddell, A. Ni, and A. Cohan, "Quantifying contamination in evaluating code generation capabilities of language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 14 116–14 137. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.761

[43] Z. Yu, Y. Wu, N. Zhang, C. Wang, Y. Vorobeychik, and C. Xiao, "Codeipprompt: Intellectual property infringement assessment of code language models," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 2023, pp. 40 373–40 389. [Online]. Available: https://proceedings.mlr.press/v202/yu23g.html