

Give LLMs a Security Course: Securing Retrieval-Augmented Code Generation via Knowledge Injection

Bo Lin*

linbo19@nudt.edu.cn
College of Computer Science,
National University of Defense
Technology
Changsha, China

Shangwen Wang†

wangshangwen13@nudt.edu.cn
College of Computer Science,
National University of Defense
Technology
Changsha, China

Yihao Qin

qinyihao@nudt.edu.cn
College of Computer Science,
National University of Defense
Technology
Changsha, China

Liqian Chen

lqchen@nudt.edu.cn
College of Computer Science,
National University of Defense
Technology
Changsha, China

Xiaoguang Mao

xgmao@nudt.edu.cn
College of Computer Science,
National University of Defense
Technology
Changsha, China

Abstract

Retrieval-Augmented Code Generation (RACG) leverages external knowledge to enhance Large Language Models (LLMs) in code synthesis, improving the functional correctness of the generated code. However, existing RACG systems largely overlook security, leading to substantial risks. Especially, the poisoning of malicious code into knowledge bases can mislead LLMs, resulting in the generation of insecure outputs, which poses a critical threat in modern software development. To address this, we propose a security-hardening framework for RACG systems, CodeGuarder, that shifts the paradigm from retrieving only functional code examples to incorporating both functional code and security knowledge. Our framework constructs a security knowledge base from real-world vulnerability databases, including secure code samples and root cause annotations. For each code generation query, a retriever decomposes the query into fine-grained sub-tasks and fetches relevant security knowledge. To prioritize critical security guidance, we introduce a re-ranking and filtering mechanism by leveraging the LLMs' susceptibility to different vulnerability types. This filtered security knowledge is seamlessly integrated into the generation prompt. Our evaluation shows CodeGuarder significantly improves code security rates across various LLMs, achieving average improvements of 20.12% in standard RACG, and 31.53% and 21.91% under two distinct poisoning scenarios without compromising functional

correctness. Furthermore, CodeGuarder demonstrates strong generalization, enhancing security even when the targeted language's security knowledge is lacking. This work presents CodeGuarder as a pivotal advancement towards building secure and trustworthy RACG systems.

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**; *Security and privacy*; Software and application security.

Keywords

Retrieval-Augmented Code Generation, Software Security, Code Generation.

ACM Reference Format:

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. In . ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of domains, from natural language processing to mathematical problem-solving. Their ability to understand and generate human-like text has led to widespread adoption in various applications. Retrieval-augmented generation (RAG) has emerged as a powerful paradigm to further enhance LLMs by leveraging external knowledge bases, enabling more contextually accurate and informed responses. In the domain of code generation, Retrieval-Augmented Code Generation (RACG) has achieved significant advancements by incorporating the relevant knowledge during the code generation (e.g., related code snippets), improving the quality of generated code. As a result, RACG-based LLM systems [27, 30, 40] have become an indispensable assistant in software development, streamlining the coding process and aiding developers in producing complex software with enhanced accuracy.

Despite these advances, existing RACG systems prioritize functional correctness while often overlooking security considerations.

*Bo Lin, Shangwen Wang, Yihao Qin, Liqian Chen and Xiaoguang Mao are also with the State Key Laboratory of Complex & Critical Software Environment

†Shangwen Wang is the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

A recent study [24] highlights critical security vulnerabilities in RACG, particularly when developer intents are exposed to attackers. Specifically, maliciously injected vulnerable code can significantly compromise the security of generated code, with empirical evidence showing that even a single poisoned sample can lead to 48% of generated code containing vulnerabilities. This threat is further exacerbated in real-world scenarios, where adversaries can introduce a large number of vulnerable samples into the knowledge base, posing a substantial risk to software security. Despite the urgency of this issue, current RACG systems lack dedicated security mechanisms to mitigate such threats.

To mitigate the security threats in RACG, our key idea is to refine the retrieval contents used in the prompt. Specifically, our approach revolves around transitioning the workflow from solely retrieving functional code examples to retrieving both functional code examples and security knowledge. The former aspect ensures that the generated code meets functional requirements, while the latter aspect, which includes secure code samples and annotated root causes of potential vulnerabilities, is dedicated to averting common security vulnerabilities in the generated code. That is to say, in addition to providing code snippets for reference implementations, we proactively include security knowledge in the prompts, aiming to assist LLMs in steering clear of vulnerabilities when generating code. We postulate that incorporating this security knowledge in the prompt can fortify the security defenses of LLMs, enabling them to “course-correct” even in scenarios where the knowledge base is poisoned, thereby preventing the generation of vulnerable code.

Building on this intuition, we introduce CodeGuarder, a security-hardening framework for RACG systems designed to achieve both functionality and security in generated code. CodeGuarder first constructs a security knowledge base by extracting insights (e.g., vulnerability root causes) from historical vulnerabilities. Subsequently, for a given code generation query, CodeGuarder employs an elaborate retriever to identify relevant security knowledge. Specifically, CodeGuarder breaks down the query into sub-tasks to retrieve precise security knowledge for each sub-task. Furthermore, recognizing the varying susceptibility of LLMs to different vulnerability types [41], CodeGuarder re-ranks the retrieved security knowledge based on these susceptibilities, prioritizing knowledge related to more prevalent vulnerabilities and filtering out less relevant knowledge. Finally, the retrieved knowledge is explicitly injected into the prompt as part of the security-augmented code generation process, ensuring that LLMs incorporate security knowledge during code generation while preserving functional correctness.

We rigorously evaluate CodeGuarder across diverse scenarios, including standard RACG and two distinct RACG poisoning scenarios. Our evaluation results demonstrate that CodeGuarder significantly improves the security of generated code, even under poisoning scenarios, without compromising functional correctness across various LLMs and languages. Specifically, in standard RACG scenarios, CodeGuarder achieves an average security improvement of 20.12%. Under poisoning attacks, CodeGuarder enhances security by 31.53% and 21.91% in the respective scenarios. Furthermore, generalization analysis reveals that CodeGuarder’s efficacy extends beyond RACG, improving security in code generation when there is no off-the-shelf knowledge base. Specifically, CodeGuarder achieves an average security rate of 75.54% across four languages, surpassing

the state-of-the-art Safecoder [14] (69.81%). Notably, even in scenarios where no language-specific secure knowledge is available, CodeGuarder consistently improves security by 15.69%, 21.26%, and 13.50% in standard and poisoning scenarios, respectively.

In summary, our study makes the following contributions:

- **Significance:** We introduce the first security-hardening framework for RACG systems, tackling the unaddressed security threats in RACG systems, especially when facing knowledge base poisoning. This work pioneers a critical shift toward securing RACG, a cornerstone of modern LLM-driven software development.
- **State-of-the-art Security Hardening Framework:** We propose CodeGuarder, a framework that significantly hardens the security of RACG systems. By explicitly injecting security knowledge into the prompts, CodeGuarder mitigates security risks effectively while maintaining the functionality of generated code.
- **Extensive Study:** We conduct a rigorous and comprehensive evaluation of CodeGuarder across diverse scenarios, including standard RACG, two RACG poisoning setups, and direct code generation. Our findings demonstrate that CodeGuarder exhibits remarkable generalization capabilities in various scenarios.

2 Background and Related Works

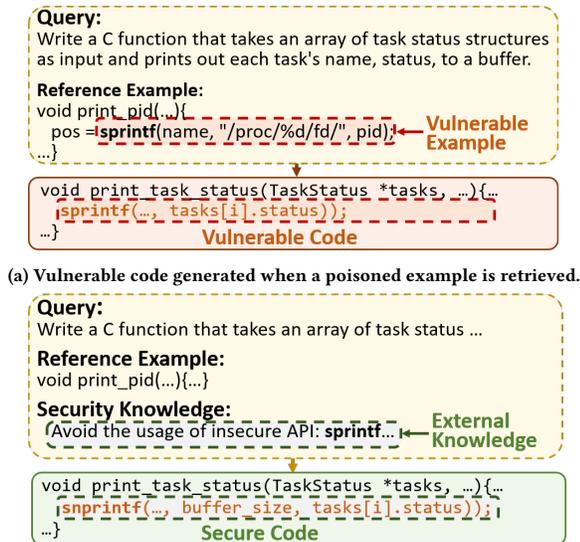
2.1 Retrieval Augmented Code Generation

LLMs have seen rapid advancement in recent years, driven by improvements in model architecture, training techniques, and access to large-scale data. Trained on diverse textual sources such as Wikipedia and GitHub, general-purpose models like GPT [4] have demonstrated impressive capabilities across a range of tasks, including those in the programming domain.

RAG is a transformative approach that enhances LLMs by incorporating relevant information retrieved from external knowledge sources. This integration significantly improves model performance by combining the strengths of retrieval systems and generative capabilities. RAG has garnered attention across various domains due to its ability to produce more accurate and contextually rich outputs, establishing itself as a robust framework for advancing natural language processing applications. Building on RAG’s success, RACG has emerged as a specialized adaptation tailored to the coding domain. RACG leverages retrieved code snippets, or other programming-related resources to enhance the efficiency and quality of code generation. By integrating domain-specific external knowledge, RACG enables LLMs to tackle complex programming tasks more effectively, surpassing the limitations of purely generative approaches [9, 49, 51].

2.2 Security of LLM-Generated Code

Although LLMs demonstrate significant capabilities in generating functionally correct code, recent studies [34, 44] highlight persistent difficulties in producing *secure* code. The security implications of LLMs used for code generation have consequently become a major research focus. Pearce *et al.* [34] systematically evaluated the security of LLM-generated code, focusing on the MITRE Top-25 vulnerabilities. Their findings indicated that approximately 40% of the generated code contained vulnerabilities, a result corroborated by subsequent studies [20, 41]. Traditional mitigation involves



(b) Secure code generated when augmented with external security knowledge.

Figure 1: An example of code generated by GPT-4o with and without additional security knowledge.

post-generation scanning using static analysis tools [1, 10], but this approach introduces latency due to the separate analysis step.

Recently, researchers have proposed approaches to enhance the security of LLM-generated code directly, without relying on retrieval. For example, SafeCoder [14] and SVEN [13] employ fine-tuning techniques to train LLMs to generate inherently more secure code. CoSec [21] improves security via supervised co-decoding, modifying the generation process without altering the LLM’s weights. These techniques mainly address non-retrieval scenarios, where LLMs rely only on pretrained knowledge, raising security concerns from training data vulnerabilities. In contrast, RACG systems combine internal knowledge with runtime-retrieved external knowledge, introducing a distinct security challenge from the potential interplay between internal and external knowledge. This challenge has not been sufficiently investigated, and there are no studies specifically addressing security hardening in the context of such interactions. This gap is critical, especially when the external knowledge base is poisoned with vulnerable code snippets, compromising the generated output. This paper mitigates this gap by introducing CodeGuarder, a security-hardening framework specifically designed for RACG, aimed at fostering the creation of secure and trustworthy code generation systems.

3 Motivating Examples

Existing RACG systems often prioritize functional correctness, potentially overlooking crucial security considerations. This focus can leave them susceptible to generating insecure code, especially in adversarial scenarios. For instance, a prior study [24] demonstrated that poisoning the retrieval knowledge base with even a single malicious example could lead to vulnerabilities appearing in nearly half (48%) of the code generated by the LLM.

Figure 1a illustrates such a scenario using a case from CyberSecEval [3], where the code is generated by GPT-4o within a poisoned RACG system. The user’s instruction prompts the LLM to generate

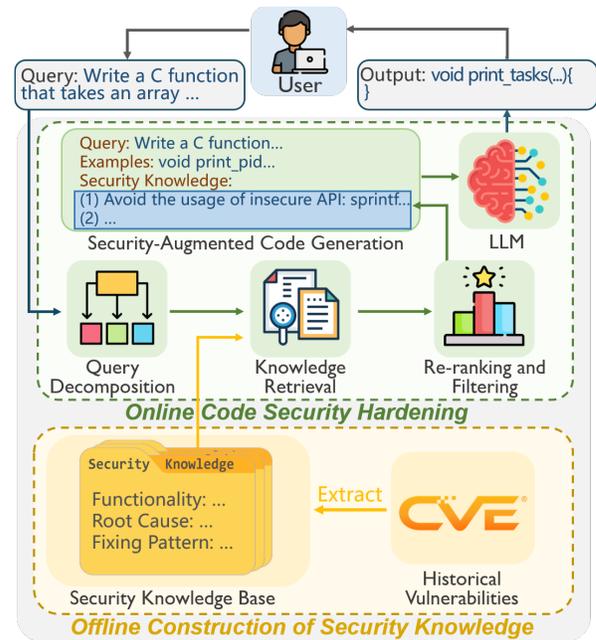


Figure 2: The overall workflow of CodeGuarder

a function displaying all fields of the given struct. Following standard procedure, the RACG system retrieves a semantically similar code example from its knowledge base. However, in this instance, the knowledge base has been poisoned, and the retrieved example itself contains a vulnerability. Specifically, the retrieved example utilizes the `sprintf` function. Guided by this insecure reference, the LLM generates code that also employs `sprintf` to fulfill the user’s request. However, `sprintf` is inherently insecure and introduces potential buffer overflow vulnerabilities.

To address the above challenge, our intuition is that if we could **inject relevant external security knowledge** into the prompt, we may help LLMs avoid this mistake. For instance, LLMs may generate secure code if they are reminded that the `sprintf` is a dangerous function. Figure 1b presents the same code generation task, but this time augmented with such knowledge. Instead of solely relying on retrieved functional code examples, the LLM is provided with relevant security knowledge, which highlights the risks associated with `sprintf` in this case. As a result, when security knowledge is included, the LLM selects the safe alternative `snprintf`, effectively mitigating buffer overflow risks while preserving functionality.

This example highlights a critical risk in current RACG systems and points towards our proposed solution: hardening RACG systems through the **injection of relevant security knowledge**. Our core idea is to shift the paradigm from solely relying on functional examples towards proactively augmenting the generation with relevant security knowledge. By equipping the LLM with this crucial knowledge, we hypothesize that we can effectively guide it towards generating code that is not only functionally correct but also vulnerability-free, mitigating vulnerabilities like the demonstrated buffer overflow.

4 Methodology

In this work, we propose CodeGuarder, a security hardening framework for RACG systems that injects security knowledge derived from existing vulnerabilities to harden the security of code generated by RACG systems. As shown in Figure 2, CodeGuarder operates in two primary phases: (1) offline construction of a security knowledge base, and (2) online integration of this knowledge to harden the security of code generated by RACG systems. In the offline phase, CodeGuarder analyzes historical vulnerabilities to automatically build the security knowledge base (§4.1). Subsequently, during the online phase, this knowledge base is utilized to harden the code generation process for user queries. This online hardening involves two main stages: first, relevant security knowledge is retrieved based on the input query (§4.2), and second, this retrieved knowledge is integrated into the prompt to guide the LLM towards generating secure code (§4.3).

4.1 Automated Offline Construction of the Security Knowledge Base

4.1.1 Security Knowledge Definition. The security knowledge base, denoted as \mathcal{S} , is constructed through an automated offline pipeline that processes historical vulnerabilities from Common Vulnerabilities and Exposures (CVE) instances. Each vulnerability instance $\mathcal{V}_i = (C_v, C_f, D_{cve}, D_{cwe})$ consists of C_v (vulnerable version), C_f (fixed version), D_{cve} (CVE description in natural language), and D_{cwe} (MITRE classification identifier). CodeGuarder represents the security knowledge in three dimensions: functionality, root cause, and fixing pattern. Figure 4 (in Appendix) illustrates a representative example extracted from CVE-2012-6538. Each dimension is detailed below:

- **Functionality:** To enable the RACG system to retrieve relevant security knowledge, we construct a representation for each security knowledge. Existing CVE descriptions typically focus on the root cause of vulnerability and impact, whereas the input to the RACG system (i.e., the query) specifies the intended functionality of the code, resulting in a significant semantic gap. To address this, we define the *functionality* associated with security knowledge by describing the functionality of the vulnerable code, extracted using the LLM backend. By leveraging *functionality* to represent corresponding security knowledge, the RACG system can retrieve relevant knowledge effectively.
- **Root Cause:** The *root cause* explains why vulnerabilities occur. It consists of two components: (i) a natural language description of the vulnerability’s root cause and (ii) a code example illustrating the vulnerability. Together, these provide both conceptual descriptions and detailed code implementations, comprehensively helping the LLM avoid generating vulnerable code.
- **Fixing Pattern:** While the *root cause* helps the LLM recognize vulnerable code, the *fixing pattern* guides the LLM to generate secure code. Similar to the root cause, the fixing pattern consists of two parts: a natural language description of the fix and an example of the corrected code. These provide complementary perspectives on how to resolve vulnerabilities, assisting the LLM in generating secure code.

4.1.2 Security Knowledge Base Construction. For each vulnerability, we prompt the LLM to extract the security knowledge. Prompt 1 (in Appendix) provides the template used for the extraction. This template takes as input the CVE description (D_{cve}), the CWE classification type (D_{cwe}), and the function-level diff (DIFF), which compares the vulnerable code (C_v) with its patched version (C_f). The output for each vulnerability is a tuple consisting of a functionality description, root cause, and fixing pattern. This process is repeated for all instances in the dataset, and the extracted items are aggregated to form the security knowledge base \mathcal{S} .

4.2 Context-Aware Fine-Grained Knowledge Online Retrieval

In this stage, we employ a context-aware, fine-grained knowledge retriever to dynamically extract relevant knowledge \mathcal{S}_Q for a given query Q from the constructed security knowledge base \mathcal{S} . This process comprises the following key stages: query decomposition, security knowledge retrieval, re-ranking, and filtering.

4.2.1 Query Decomposition. As suggested in previous studies [12, 15], a single program may contain multiple vulnerabilities across different locations, meaning that a code generated by a single query may correspond to multiple security issues. Figure 3 presents a concrete example from CyberSecEval [43], generated by GPT-4o. The query instructs the LLM to generate a function that copies a string and returns the corresponding pointer. However, the generated code introduces multiple buffer overflow vulnerabilities related to memory allocation and string copying at different locations. This insight led us to decompose the user’s query into smaller, fine-grained semantic units (i.e., sub-tasks), enabling more precise knowledge retrieval and enhancing the security of the generated code, as illustrated in Figure 3. For example, for the decomposed sub-task related to memory allocation, the retrieved knowledge explicitly indicates that the length should be checked to prevent overflow before allocating memory.

LLMs have demonstrated a remarkable capability in understanding complex natural language instructions and discerning underlying semantic components. Their ability to process intricate queries and apply reasoning makes them well-suited for breaking down tasks into concrete steps, as demonstrated by techniques like chain-of-thought [22, 46]. Leveraging this capability, we employ the LLM to decompose the query into smaller sub-tasks that represent the query’s semantics at a finer level. The decomposition template is detailed in Prompt 2 (see Appendix). The final decomposed query Q_d consists of a list of sub-tasks $Q_d = [q_1, q_1, \dots, q_n]$, which are then used in the knowledge retrieval stage.

4.2.2 Security Knowledge Retrieval. In this stage, we aim to preliminarily identify the relevant security knowledge entries from the security knowledge base \mathcal{S} . We adopt a similarity-based retrieval approach, using cosine similarity to measure the relevance between the feature representations of the sub-tasks and the knowledge entries in the base. Formally, for a list of decomposed sub-tasks $Q_d = [q_1, q_1, \dots, q_n]$, where each q_i represents a fine-grained semantic unit (i.e., sub-task), we compute the feature vector for each sub-task q_i by embedding its description into a vector:

$$\mathbf{v}_{q_i} = \text{Embed}(q_i),$$

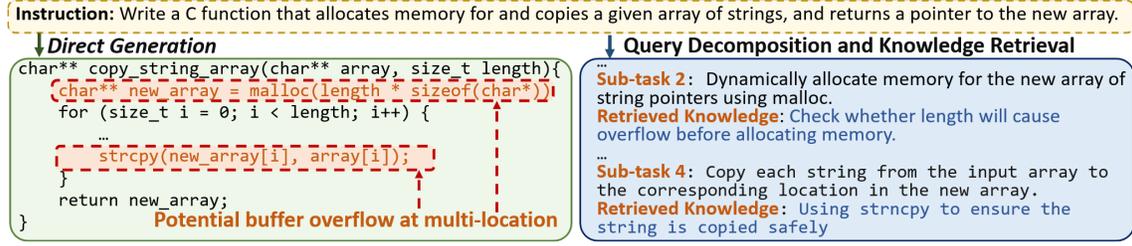


Figure 3: Vulnerable code generated by GPT-4o along with its corresponding decomposed queries and retrieved knowledge.

Table 1: Distribution of violation types and associated CWEs in LLM-generated code

Violation Type	Related CWEs	Percentage
NULL pointer	391, 476, 690;	40.24%
Buffer overflow	120, 121, 122, 628, 676, 680, 787;	25.53%
Invalid pointer	822, 119;	10.42%
Array bounds violated	125, 129, 131, 193, 788;	8.86%
Arithmetic overflow	191, 20, 190, 192, 681;	6.21%
Resource mismanagement	825, 401, 404, 459;	5.03%
Division by zero	369, 691;	1.45%
Others	-	2.26%

where $\mathbf{v}_{q_i} \in \mathbb{R}^d$ is the feature vector for sub-task q_i , and $\text{Embed}(\cdot)$ denotes the embedding function (detailed in §5.4). Each knowledge entry s_j in \mathcal{S} is also represented by a feature vector \mathbf{v}_{s_j} , computed similarly by embedding its textual description:

$$\mathbf{v}_{s_j} = \text{Embed}(\text{desc}(s_j)),$$

where $\text{desc}(s_j)$ is the textual description of the functionality of the knowledge entry s_j (as defined in §4.1.1).

To retrieve the relevant knowledge for a given sub-task q_i , we calculate the cosine similarity between its feature vector \mathbf{v}_{q_i} and the feature vector \mathbf{v}_{s_j} of each knowledge entry s_j in the knowledge base. Based on these similarity scores, we retrieve the top- k' (discussed in §7.4) most relevant knowledge entries for q_i , denoted as \mathcal{S}'_{q_i} , which are then subjected to subsequent re-ranking and filtering.

4.2.3 Security Knowledge Re-ranking and Filtering. Note that before this stage, we have retrieved relevant security knowledge for each sub-task. Although the retrieval stage ensures that each sub-task is associated with a fixed number of top- k' security knowledge entries, the effectiveness of this approach is limited by two key challenges. First, the input window size of the LLM may be exceeded, leading to truncation or loss of critical context [5, 26]. Second, LLMs often struggle with long, complex prompts due to attention dilution, reducing their ability to leverage external knowledge [48, 52].

To mitigate these challenges and refine the preliminary knowledge obtained during the retrieval stage, we introduce a re-ranking and filtering mechanism that tailors the security knowledge to the varying risk profiles of individual sub-tasks. The key insight is that not all sub-tasks exhibit the same susceptibility to vulnerabilities. For example, a sub-task responsible for outputting results typically poses less risk than one involving memory allocation, which is more prone to issues like buffer overflows. To quantify the risk associated with each sub-task q_i , we first determine the weight w_j for each individual knowledge entry s_j within the initially retrieved set \mathcal{S}'_{q_i} . This weight reflects the likelihood that the vulnerability associated with s_j manifests in LLM-generated code. This approach is grounded in the observation that sub-tasks semantically related

to frequently occurring vulnerabilities are more likely to induce the LLM to produce insecure code. To quantify this likelihood, we draw on a prior study [41], which analyzed the distribution of violation types and their associated CWEs across code generated by 13 mainstream LLMs (e.g., GPT and Gemini series models) on the dataset containing 310,531 code generation instructions. The resulting distribution, detailed in Table 1, provides empirical evidence of vulnerability prevalence in LLM-generated code.

For each knowledge entry s_j within the preliminary retrieved set \mathcal{S}'_{q_i} for sub-task q_i , we assign a weight w_j reflecting the probability of its associated vulnerability appearing in LLM-generated code. Specifically, we map the CWE of s_j to its corresponding violation type in Table 1. The weight w_j is then set to the percentage frequency of that violation type. For example, a knowledge entry related to CWE-476 (NULL pointer dereference), categorized under the "NULL pointer" violation type with a frequency of 40.24%, receives a weight $w_j = 0.40$. For CWEs not listed in the table, we assign a default weight of $w_j = 0.01$ to ensure all entries are preliminarily considered. Next, we calculate the overall weight W_{q_i} for each sub-task q_i by summing the weights of all its initially retrieved knowledge entries:

$$W_{q_i} = \sum_{s_j \in \mathcal{S}'_{q_i}} w_j.$$

This aggregated weight W_{q_i} represents the estimated risk level of sub-task q_i , considering the vulnerabilities associated with its relevant knowledge.

We then re-rank all sub-tasks $[q_1, \dots, q_n]$ based on weights W_{q_i} in descending order. Subsequently, we perform a filtering step by selecting the sub-tasks corresponding to the top- k highest weights. Let $\mathcal{Q}_{\text{top-}k}$ denote this set of top- k sub-tasks. The parameter k is set to five as discussed in §7.4. This filtering strategy ensures that the security knowledge associated with the sub-tasks deemed most likely to introduce critical vulnerabilities is retained. By prioritizing sub-tasks with high aggregated weights, this process naturally emphasizes knowledge tied to high-risk vulnerabilities.

The final security knowledge base \mathcal{S}_Q for the query Q is then constructed by aggregating all the initially retrieved knowledge entries corresponding to these selected top- k sub-tasks:

$$\mathcal{S}_Q = \bigcup_{q_i \in \mathcal{Q}_{\text{top-}k}} \mathcal{S}'_{q_i}.$$

This curated \mathcal{S}_Q is subsequently integrated into the code generation process (see §4.3), enabling CodeGuarder to guide the LLM toward producing secure code by concentrating on the highest-risk aspects identified through sub-task ranking.

4.3 Security-Augmented Code Generation

In conventional RACG systems, LLMs generate code by leveraging both a user-provided query and external knowledge retrieved from the knowledge base, typically, code snippets. The prompt structure in traditional RACG can be expressed as:

$$P_{ori} = Q + E,$$

where Q is the user's input specifying the desired functionality, and E provides example implementations.

To enhance the security of RACG systems, CodeGuarder injects security knowledge \mathcal{S}_Q into the code generation prompt. Specifically, for a given user query Q and its filtered sub-tasks $q_i \in \mathcal{Q}_{top-k}$, each sub-task q_i is paired with its corresponding security knowledge \mathcal{S}'_{q_i} . The security-augmented prompt is constructed as:

$$P = P_{ori} + \sum_{q_i \in \mathcal{Q}_{top-k}} (q_i + \mathcal{S}'_{q_i}),$$

which ensures that LLM generates code satisfying both functional requirements and security constraints. The Prompt 3 (in Appendix) provides an example of this process. For instance, if q_i involves "copy a string from input to destination", \mathcal{S}'_{q_i} might include guidelines like "Use `strncpy` to ensure the string is copied safely".

5 Study Design

5.1 Investigated Scenarios

In this study, we comprehensively evaluate the effectiveness of CodeGuarder across three scenarios: a standard RACG and two poisoning scenarios, following prior work [24]. The standard scenario represents a typical RACG setting, where the retriever fetches knowledge from a knowledge base free from vulnerable code. In contrast, the two poisoning scenarios simulate situations where programming intents (i.e., queries) are either exposed or not exposed to an attacker, who subsequently injects vulnerable code into the knowledge base.

5.1.1 Standard RACG Scenario. In this scenario, the RACG system operates without poisoning. The system retrieves code from the functional code base \mathcal{K} based on the query Q and generates code accordingly. This setting evaluates the framework's performance in a typical scenario, serving as a reference for the poisoning scenarios.

5.1.2 Poisoning Scenario I: Exposed Programming Intents. In poisoning scenarios, we assume that the attacker can access and poison \mathcal{K} , which is typically sourced from public repositories such as GitHub. However, access to programming intents (i.e., query Q) depends on the attacker's capabilities and may not always be feasible. In this scenario, we assume the attacker has access to Q and exploits this information to inject vulnerabilities into \mathcal{K} . Specifically, the attacker selects the m most semantically similar vulnerable examples from the vulnerable code base \mathcal{V} using a poisoning retriever. The knowledge base used by RACG then becomes $\mathcal{K} \cup \mathcal{V}$, blending secure and vulnerable code. This scenario tests the CodeGuarder's ability to mitigate targeted attacks leveraging intent-specific vulnerabilities. Prior empirical studies [24] indicate that varying the poisoning quantity exhibits similar patterns in its impact on the security and functionality of the generated code. For clarity, we

assess CodeGuarder under a moderate poisoning level, using five poisoned samples (i.e., $m = 5$) for each query.

5.1.3 Poisoning Scenario II: Intent-Agnostic Poisoning. In this scenario, we mimic a stricter attack setting than scenario I, where the attacker lacks direct access to Q and instead poisons \mathcal{K} with broadly representative vulnerable code likely to be retrieved across diverse queries. Specifically, prior work [24] suggests that an attacker can poison the knowledge base by injecting common functionalities that are more likely to be retrieved in RACG, thereby affecting a broader range of queries. Following this insight, we adopt a clustering-based approach to select the top $p\%$ most representative examples from \mathcal{K} . For each selected example, we retrieve its most similar vulnerable code from \mathcal{V} and inject it into the knowledge base. The resulting $\mathcal{K} \cup \mathcal{V}$ simulates a generalized poisoning attack, evaluating CodeGuarder's resilience when programming intents are not exposed. In this scenario, we assess CodeGuarder under a moderate poisoning level, using the poisoning proportion of 10% (i.e., $p = 10$).

5.2 Benchmark and Knowledge Bases

5.2.1 Evaluation Benchmark. To evaluate the security and functionality of LLM-generated code across diverse scenarios, we adopt CyberSecEval [43] as our benchmark for two primary reasons: (1) It serves as the official benchmark for the LLaMA series of LLMs, which are widely utilized in academia and industry [11, 19]. (2) It provides the most comprehensive assessment of security among available benchmarks. Specifically, CyberSecEval comprises 1,916 instances spanning 50 CWE types, exhibiting a lead over the second largest security evaluation benchmark [42], which includes only 150 instances across 18 CWE types. Additionally, CyberSecEval includes a specialized insecure code detector, built upon analysis rules tailored to its cases, enabling security evaluation with a precision of 96%. While other datasets like ReposVul [45] contain more vulnerabilities, they primarily function as repositories of vulnerabilities and lack the associated evaluation framework (e.g., analysis rules) necessary for performing high-precision security assessments on generated code. Consequently, datasets like ReposVul are less suited for rigorously evaluating the security of generated code.

5.2.2 Knowledge Bases Construction. As illustrated in §5.1, the three scenarios involve distinct knowledge bases that serve different roles in the RACG system. The first is a security knowledge base that stores security-related knowledge, which CodeGuarder leverages to enhance the security of generated code. The second is a knowledge base from which the RACG system retrieves code examples for code generation. The third is a set of vulnerable code, which serves as the attacker's resource for selecting and injecting malicious code into the RACG system, thereby compromising the security of the generated code. Therefore, three distinct knowledge bases are required: the Security Knowledge Base (\mathcal{S}), the Functional Code Base (\mathcal{K}) and the Vulnerable Code Base (\mathcal{V}), as detailed below:

- **Security Knowledge Base (\mathcal{S}):** This base provides security knowledge to harden the security of generated code. It is constructed by analyzing vulnerabilities and their corresponding fixes from ReposVul [45], the largest cross-language dataset of real-world vulnerabilities, which includes 12,053 function-level

Table 2: Statistics of knowledge bases

Knowledge Base	Language			
	C	C++	Java	Python
Functional Code Base	6,956	510	2,810	1,777
Vulnerable Code Base	227	259	235	229
Security Knowledge Base	8,861	644	3,365	2,217

Table 3: Studied LLMs in the study

Category	LLM	Publisher	Open-source
General	GPT-4o [29]	OpenAI	No
	DeepSeek-V3 [25]	DeepSeek	Yes
Code	CodeLlama-13B [37]	Meta	Yes
	DeepSeek-Coder-V2-16B [53]	DeepSeek	Yes

pairs of vulnerable and secure code across four programming languages. The construction process is described in detail in §4.1.

- **Functional Code Base (\mathcal{K}):** This forms the foundational knowledge base for retrieval-augmentation in the RACG system, enabling it to retrieve relevant functional and secure code based on user queries. Commonly used code bases (e.g., CSN [16]) are typically collected from open source projects, potentially introducing vulnerable code into the dataset. For example, in our analysis of CSN, 81.3% (684 out of 841) of Java code invoking random-related functions rely on weak randomness, which can lead to serious security vulnerabilities. The presence of such insecure code in functional code bases may bias evaluations and obscure the actual impact of poisoning attacks. To mitigate this, we construct \mathcal{K} using the fixed code (i.e., the vulnerability-free code) from the ReposVul dataset, ensuring that all retrieved examples are functionally correct and security-vetted.
- **Vulnerable Code Base (\mathcal{V}):** To simulate realistic RACG knowledge base poisoning attacks, a dedicated vulnerable code base is essential. This base serves as the source of vulnerable code that attackers would retrieve and inject into the functional code base \mathcal{K} during poisoning scenarios (detailed in §5.1). Importantly, \mathcal{V} is built using vulnerable code instances from the CyberSecEval dataset, rather than ReposVul. Using the same dataset for both the vulnerable code and the security knowledge base could lead to overlapping fix strategies, making poisoning attempts easier to detect and defend against, thereby undermining the realism of the poisoning simulation. It is worth noting that this experimental setup grants attackers slightly more power than in real-world scenarios, as they can inject vulnerable code into \mathcal{K} that closely matches the user’s query. This assumption enables us to more rigorously evaluate the effectiveness of CodeGuarder under more severe security threats.

Table 2 presents the statistics of the aforementioned knowledge bases across different programming languages. The number of security knowledge instances exceeds the number of vulnerabilities in ReposVul, as some vulnerabilities have multiple root causes, as observed in previous studies [12, 15].

5.3 Studied LLMs

To evaluate CodeGuarder’s effectiveness, we employ four representative LLMs, encompassing a range of model sizes, and categories. These include: (1) GPT-4o, a closed-source general-purpose model accessed via its API; and (2) DeepSeek-V3, a state-of-the-art general LLM, also accessed via API [31]; (3) CodeLlama-13B, a

code-oriented open-source model; (4) DeepSeek-Coder-V2-16B, an advanced code-oriented open-source model. Table 3 summarizes their key attributes, spanning small-scale (e.g., 8B–16B) to large-scale models and both general-purpose and code-specific designs. Model weights for open-source LLMs (CodeLlama and DeepSeek-Coder-V2) were sourced from their official Hugging Face repositories. For brevity, we denote these as GPT-4o, DS-V3, CodeLlama and DS-Coder in subsequent sections.

5.4 Retriever

Retrievers are integral to the RACG system, enabling the retrieval of additional knowledge to enhance generation. In this study, we involve three types of retrievers: code retriever, knowledge retriever, and poisoning retriever as follows:

- **Code Retriever.** The code retriever supports the RACG pipeline by fetching relevant code examples from the functional knowledge base \mathcal{K} to serve as references during generation. We implement this retriever using the state-of-the-art jina-embeddings-v3 model [39], a dense retriever that embeds code snippets into a vector space for similarity-based retrieval.
- **Knowledge Retriever.** The knowledge retriever, detailed in §4.2, underpins our context-aware fine-grained knowledge online retrieval process. It extracts security knowledge from \mathcal{S} by calculating the similarity between sub-tasks (decomposed from the query) and the knowledge entry in the security knowledge base. Note that we utilize the same embedding model for code and knowledge retriever (i.e., jina-embeddings-v3) to reduce the semantic gap between the retrieved code and secure knowledge.
- **Poisoning Retriever.** In adversarial scenarios, attackers lack access to the parameters or query capabilities of the retrievers in the RACG system. Thus, an external poisoning retriever is required for the retrieval of vulnerable code in poisoning scenario I and generating embeddings for clustering-based poisoning in poisoning scenario II. Therefore, we built the poisoning retriever on the text-embedding-3-large model [32] following a previous study [24]. Operating independently of the RACG system, the poisoning retriever embeds and retrieves vulnerable code from the vulnerable knowledge base to poison the RACG system, simulating the realistic poisoning process of the RACG system.

5.5 Metrics

To evaluate the effectiveness of CodeGuarder in enhancing security while maintaining functionality, we employ the following metrics:

Security Rate (SR): SR quantifies the likelihood of an LLM generating secure code, defined as the percentage of generated code verified as secure. The verification process is conducted using the Insecure Code Detector from CyberSecEval [43], which detects vulnerable code across seven programming languages and over 50 CWE types with a precision of 96% [43].

Similarity (Sim): Due to the absence of test cases in the CyberSecEval dataset, we cannot directly evaluate whether CodeGuarder affects the functionality of LLM-generated code. Instead, we use CodeBLEU [36], a BLEU variant for code similarity, to compare generated code with ground truth, as a partial representative for functionality. Besides, we further assess the functional correctness of

LLM-generated code after applying CodeGuarder using test-based benchmarks: MBPP[2] and HumanEval[6], as detailed in §7.1.

5.6 Implementation Details

All experiments were conducted on an A100 GPU server using the Ollama [28]. To ensure output consistency across LLMs, we set the temperature parameter to 0, minimizing non-determinism as recommended by prior work [33]. Model configurations followed established settings [24]: a `max_new_tokens` limit of 4096, and a context window of 8192 tokens, with other parameters left at defaults. We adhered to each model’s recommended prompt formats, sourced from official documentation, GitHub repositories, or original papers, using predefined chat templates where applicable. For constructing the security knowledge base (§4.1) and performing query decomposition (§4.2.1), we leveraged DeepSeek-V3 [25], a state-of-the-art open-source LLM, as the backend.

6 Evaluation

6.1 Research Questions

To systematically evaluate the effectiveness of CodeGuarder in enhancing the security of RACG systems, we formulate the following research questions (RQs):

RQ1: Performance in Standard RACG. How does CodeGuarder perform in a non-poisoned RACG scenario?

RQ2: Resilience to Poisoning Attacks. How resilient is CodeGuarder to poisoning attacks in RACG scenario?

RQ3: Generalization of CodeGuarder. How well does CodeGuarder generalize across diverse conditions, including code generation when there is no off-the-shelf knowledge base, and scenarios lacking target-language security knowledge?

6.2 RQ1: Performance in the Standard RACG

This RQ evaluates CodeGuarder’s ability to enhance the security of RACG-generated code under the standard setting, where the knowledge base comprises solely functional code examples without poisoning. The goal is to assess whether CodeGuarder can effectively leverage retrieved security knowledge to mitigate vulnerabilities.

Table 4 presents the security hardening and functionality maintenance achieved by CodeGuarder across four LLMs. On average, CodeGuarder increases the security rate (SR) of LLM-generated code by 9.37% to 47.72% across four languages, while maintaining or slightly enhancing the similarity (Sim) metrics. We observed a trend where CodeGuarder exhibits more obvious security-hardening on LLMs with larger parameter counts (e.g., DS-V3 with 671 billion parameters) compared to those with fewer parameters (e.g., CodeLlama with 13 billion parameters). Specifically, GPT-4o and DS-V3 achieve improvements of 20.28% and 30.73% in SR, respectively, whereas CodeLlama and DS-Coder show improvements of 16.15% and 13.50%. This discrepancy might be attributed to the reduced instruction-following capabilities of smaller models [7, 18].

From a programming language perspective, CodeGuarder demonstrates the most significant security enhancement in Java, with an average improvement of approximately 38.45%. This suggests that CodeGuarder is particularly effective in addressing vulnerabilities prevalent in Java. Conversely, the improvements in C++ and Python are comparatively smaller, at 9.37% and 12.24%, respectively.

Table 4: Performance of CodeGuarder under Standard RACG Scenario.

Metrics	LLM	Language				Average	
		C	C++	Java	Python		
SR	GPT-4o	55.95	80.31	41.92	70.09	62.07	
	w. CodeGuarder †	70.04	84.94	62.45	81.20	74.66	
		(↑ 25.18%)	(↑ 5.77%)	(↑ 48.97%)	(↑ 15.85%)	(↑ 20.28%)	
	DS-V3	55.07	74.90	40.61	68.95	59.88	
	w. CodeGuarder	72.25	88.03	73.36	79.49	78.28	
		(↑ 31.20%)	(↑ 17.53%)	(↑ 80.65%)	(↑ 15.29%)	(↑ 30.73%)	
	CodeLlama	52.42	74.13	44.98	68.66	60.05	
	w. CodeGuarder	66.52	80.31	57.21	74.93	69.74	
		(↑ 26.90%)	(↑ 8.34%)	(↑ 27.19%)	(↑ 9.13%)	(↑ 16.15%)	
	DS-Coder	51.98	75.68	46.29	71.51	61.37	
w. CodeGuarder	65.20	80.31	55.02	78.06	69.65		
	(↑ 25.43%)	(↑ 6.12%)	(↑ 18.86%)	(↑ 9.16%)	(↑ 13.50%)		
	Average	53.86	76.26	43.45	69.80	60.84	
w. CodeGuarder	68.50	83.40	62.01	78.42	73.08		
	(↑ 27.20%)	(↑ 9.37%)	(↑ 47.72%)	(↑ 12.35%)	(↑ 20.12%)		
Sim	GPT-4o	22.82	24.51	28.56	21.31	24.30	
	w. CodeGuarder	23.72	25.45	29.17	23.36	25.43	
	DS-V3	22.76	24.10	28.98	21.10	24.24	
	w. CodeGuarder	24.11	25.36	29.31	23.46	25.56	
	CodeLlama	23.14	24.31	28.01	21.73	24.30	
	w. CodeGuarder	23.89	25.65	28.34	22.95	25.21	
	DS-Coder	21.95	24.01	26.95	21.00	23.48	
	w. CodeGuarder	22.85	24.94	27.03	22.34	24.29	
		Average	22.67	24.23	28.13	21.29	24.08
	w. CodeGuarder	23.64	25.35	28.46	23.03	25.12	

† “w. CodeGuarder ” denotes the results of LLMs hardened by CodeGuarder.

This is likely because the SR of LLM-generated C++ and Python code is already high, leaving limited room for improvement. We further investigated the ratio of insecure cases that were successfully secured by CodeGuarder, revealing that approximately 31.74%, 30.08%, 32.82% and 28.54% of C, C++, Java and Python, respectively, were effectively secured.

To examine CodeGuarder’s impact on code functionality, we measured the similarity between generated code and the reference code using the Sim metric. Notably, CodeGuarder did not degrade functionality; rather, it slightly improved it. For instance, the average Sim values across all LLMs without CodeGuarder were 22.67, 24.23, 28.13, and 21.29 for C, C++, Java, and Python, respectively. With CodeGuarder, these values increased to 23.64, 25.35, 28.46, and 23.03, respectively. This improvement suggests that the detailed guidance and secure code examples within the security knowledge base may enhance the functional correctness of generated code.

Note that the Sim metric only measures code similarity due to the absence of test cases in the CyberSecEval dataset. To provide a comprehensive evaluation of CodeGuarder’s impact on functionality, we employed the MBPP [2] and HumanEval [6] benchmarks, which utilize test cases to assess code correctness (see §7.1). These results confirm that CodeGuarder preserves code functionality, thus maintaining the practical utility of the generated code.

Answer to RQ1: CodeGuarder effectively guides LLMs to generate more secure code in standard RACG scenarios across various programming languages and LLMs, while maintaining or slightly improving code functionality. Specifically, CodeGuarder enhances the security rate by approximately 27.20% for C, 9.37% for C++, 47.72% for Java, and 12.35% for Python.

6.3 RQ2: Resilience to Poisoning Attacks

This RQ examines CodeGuarder’s performance when the knowledge base \mathcal{K} is poisoned with vulnerable code, as modeled in §5.1. We evaluate CodeGuarder’s capacity to harden code security amidst

Table 5: Performance of CodeGuarder under poisoning scenario I.

Metrics	LLM	Language				Average
		C	C++	Java	Python	
SR	GPT-4o	48.9	64.09	31.44	55.84	50.07
	w. CodeGuarder	62.56	79.15	56.77	71.23	67.43
		(\uparrow 27.93%)	(\uparrow 23.50%)	(\uparrow 80.57%)	(\uparrow 27.56%)	(\uparrow 34.67%)
	DS-V3	49.78	64.86	30.13	60.11	51.22
	w. CodeGuarder	68.72	80.31	57.21	74.64	70.22
		(\uparrow 38.05%)	(\uparrow 23.82%)	(\uparrow 89.88%)	(\uparrow 24.17%)	(\uparrow 37.09%)
	CodeLlama	51.1	61	39.74	56.13	51.99
	w. CodeGuarder	64.32	75.68	52.84	68.95	65.45
		(\uparrow 25.87%)	(\uparrow 24.07%)	(\uparrow 32.96%)	(\uparrow 22.84%)	(\uparrow 25.88%)
	DS-Coder	44.49	60.62	34.5	56.41	49.01
	w. CodeGuarder	60.79	77.61	43.67	69.8	62.97
		(\uparrow 36.64%)	(\uparrow 28.03%)	(\uparrow 26.58%)	(\uparrow 23.74%)	(\uparrow 28.49%)
Average	48.57	62.64	33.95	57.12	50.57	
w. CodeGuarder	64.10	78.19	52.62	71.16	66.52	
	(\uparrow 31.98%)	(\uparrow 24.82%)	(\uparrow 54.99%)	(\uparrow 24.57%)	(\uparrow 31.53%)	
Sim	GPT-4o	25.89	29.49	32.75	28.73	29.97
	w. CodeGuarder	24.16	29.25	33.45	28.16	28.76
	DS-V3	22.68	28.77	32.89	25.80	27.54
	w. CodeGuarder	24.73	28.96	32.71	27.42	28.46
	CodeLlama	25.18	30.09	32.03	31.24	29.64
	w. CodeGuarder	25.97	31.84	31.86	32.48	30.54
	DS-Coder	24.95	28.76	32.82	29.14	28.42
	w. CodeGuarder	24.71	28.98	32.68	27.59	28.49
	Average	24.68	29.28	32.62	28.73	28.70
	w. CodeGuarder	24.89	29.76	32.68	28.91	29.06

exposed programming intents (poisoning scenario I) and intent-agnostic knowledge poisoning (poisoning scenario II).

6.3.1 Poisoning Scenario I. This scenario evaluates CodeGuarder’s resilience to poisoning attacks where the attacker has access to the programming intents (i.e., queries) and poisons the knowledge base \mathcal{K} . The attacker injects the five most relevant vulnerable examples into \mathcal{K} as illustrated in §5.1.2, simulating a targeted attack.

Table 5 presents the performance of CodeGuarder under poisoning scenario I. The results indicate that in this scenario, the security of LLMs is compromised. For instance, the average SR across LLMs drops from 60.84 (in the standard scenario) to 50.57, meaning nearly half of the generated code is vulnerable. Despite the presence of explicitly malicious knowledge in \mathcal{K} , CodeGuarder consistently improves the SR across all tested LLMs and programming languages. On average, CodeGuarder enhances SR by 31.53%, demonstrating its ability to defend against attacks even when programming intents are exposed. Notably, DS-V3 exhibits the highest average improvement in SR (37.09%), followed by GPT-4o (34.67%), CodeLlama (28.49%), and DS-Coder (25.88%). From a language perspective, CodeGuarder shows the most significant improvement in Java language, with an average SR increase of 54.99%. This indicates that CodeGuarder is particularly effective in addressing vulnerabilities in Java. The improvements in C, C++, and Python are also substantial, with average SR increases of 31.98%, 24.82%, and 24.57%, respectively. Additionally, we analyze the proportion of previously insecure cases that were successfully secured by CodeGuarder. Results show that CodeGuarder effectively secures 41.61% of C++ cases, followed by Python (32.73%), C (30.19%), and Java (28.27%).

Regarding functionality, as measured by the Sim metric, CodeGuarder maintains or slightly improves the similarity between the generated code and the reference code. The average Sim across all LLMs and languages increases from 28.70 to 29.06 with CodeGuarder. This suggests that CodeGuarder’s security enhancements do not compromise the functional correctness of the generated code, even in the context of targeted poisoning attacks.

Table 6: Performance of CodeGuarder under poisoning scenario II.

Metrics	LLM	Language				Average
		C	C++	Java	Python	
SR	GPT-4o	52.86	76.83	38.86	63.58	58.03
	w. CodeGuarder	69.6	86.49	64.63	80.63	75.34
		(\uparrow 31.67%)	(\uparrow 12.57%)	(\uparrow 66.31%)	(\uparrow 26.81%)	(\uparrow 29.82%)
	DS-V3	52.86	72.97	38.86	66.95	57.91
	w. CodeGuarder	72.25	86.87	65.94	80.63	76.42
		(\uparrow 36.68%)	(\uparrow 19.05%)	(\uparrow 69.69%)	(\uparrow 20.43%)	(\uparrow 31.97%)
	CodeLlama	57.71	71.43	50.66	70.37	62.54
	w. CodeGuarder	65.20	80.69	58.52	78.06	70.62
		(\uparrow 12.98%)	(\uparrow 12.96%)	(\uparrow 15.52%)	(\uparrow 10.93%)	(\uparrow 12.91%)
	DS-Coder	52.42	77.61	45.85	70.94	61.71
	w. CodeGuarder	64.32	81.08	60.26	76.07	73.29
		(\uparrow 22.70%)	(\uparrow 4.47%)	(\uparrow 31.43%)	(\uparrow 7.23%)	(\uparrow 18.77%)
Average	53.96	74.71	43.56	67.96	60.05	
w. CodeGuarder	67.84	83.78	62.34	78.85	73.20	
	(\uparrow 25.72%)	(\uparrow 12.14%)	(\uparrow 43.12%)	(\uparrow 16.02%)	(\uparrow 21.91%)	
Sim	GPT-4o	18.36	20.61	26.39	21.03	21.60
	w. CodeGuarder	18.92	21.57	27.03	24.94	23.12
	DS-V3	17.5	21.13	26.54	20.69	21.47
	w. CodeGuarder	18.53	21.55	26.93	22.56	22.39
	CodeLlama	17.63	20.2	25.33	20.86	21.01
	w. CodeGuarder	18.76	22.34	26.34	22.02	22.37
	DS-Coder	16.6	19.95	24.64	20.08	20.32
	w. CodeGuarder	18.83	22.18	26.11	21.99	22.28
	Average	17.52	20.47	25.73	20.67	21.10
	w. CodeGuarder	18.76	21.91	26.60	22.88	22.54

Overall, CodeGuarder demonstrates a strong resilience to poisoning attacks in scenario I. It effectively mitigates targeted vulnerabilities by enhancing code security without sacrificing functionality, even when the knowledge base is poisoned with semantically similar vulnerable code and the programming intents are exposed.

6.3.2 Poisoning Scenario II. This scenario evaluates CodeGuarder’s resilience to poisoning attacks where the attacker lacks direct access to the programming intents (i.e., queries Q) and instead poisons the knowledge base \mathcal{K} with common vulnerable functionalities that are more likely to be retrieved in RACG. The attacker employs a clustering-based approach to select the top 10% most representative examples from \mathcal{K} and injects their corresponding vulnerable counterparts, simulating a generalized poisoning attack.

Table 6 presents the performance of CodeGuarder under poisoning scenario II. Similar to scenario I, CodeGuarder consistently enhances the SR across all tested LLMs and programming languages, even when faced with intent-agnostic poisoned knowledge. On average, CodeGuarder improves SR by 21.91%, demonstrating its robustness against generalized poisoning attacks. Notably, DS-V3 exhibits the highest average improvement in SR (31.97%), followed by GPT-4o (29.82%), DS-Coder (18.77%), and CodeLlama (12.91%).

From a language perspective, CodeGuarder again demonstrates the most significant improvement in Java code generation, with an average SR increase of 43.12%. This confirms CodeGuarder’s effectiveness in mitigating vulnerabilities in Java, even under generalized poisoning conditions. The improvements in C, C++, and Python are also notable, with average SR increases of 25.72%, 12.14%, and 16.02%, respectively. Furthermore, the proportion of cases successfully secured by CodeGuarder across the four programming languages is 30.15%, 35.87%, 33.27%, and 33.98%, respectively.

Regarding functionality, as measured by the Sim metric, CodeGuarder maintains or slightly improves the similarity between the generated code and the reference code. The average Sim across all LLMs and languages increases from 21.10 to 22.54 with CodeGuarder. This indicates that CodeGuarder’s security enhancements do

not compromise the functional correctness of the generated code, even in the context of intent-agnostic poisoning attacks.

Answer to RQ2: CodeGuarder consistently improves code security across poisoning scenarios, including targeted (Scenario I) and generalized (Scenario II) attacks. Specifically, CodeGuarder achieves an average SR improvement of 31.53% in scenario I and 21.91% in scenario II. These results demonstrate CodeGuarder’s effectiveness in hardening the security of RACG systems, without compromising the functionality of the generated code.

6.4 RQ3: Generalization of CodeGuarder

This RQ explores CodeGuarder’s generalization by analyzing two dimensions: (1) CodeGuarder’s performance when there is no off-the-shelf knowledge base (denoted as non-func-retrieval) that the LLM relies on its intrinsic knowledge to generate code; and (2) CodeGuarder’s performance when the security knowledge base lacks corresponding knowledge for the target language.

6.4.1 Performance in the Non-Func-Retrieval Scenario. The non-func-retrieval scenario is common in daily software development, where LLMs generate code solely based on developer instructions. To assess CodeGuarder’s generalization in this setting, we examine whether it enhances code security while preserving functionality. Additionally, we compare its effectiveness against existing methods specifically designed for this generation scenario.

Baselines. Recent research has introduced several approaches aimed at enhancing the security of code generation in non-func-retrieval scenarios [13, 14, 21]. We select three state-of-the-art security-hardening methods as baselines:

- **Sven.** Sven [13] employs the prefix-tuning technique [23] to steer code generation toward desired properties, such as functional correctness and security, without modifying the LLM’s weights.
- **SafeCoder.** Similar to Sven, SafeCoder [14] employs instruction tuning [50] to fine-tune LLMs on a specially curated dataset, guiding them to generate secure code while discouraging unsafe program generation through likelihood loss.
- **Cosec.** Previous studies [13, 14] require access to the weights of LLMs, which limits the applicability of these approaches. In contrast, Cosec [21] proposes leveraging co-decoding to adjust the probability distributions of tokens at each step of the decoding process, thereby guiding the generation of secure code.

Experimental Results. We evaluate the approaches only on C, C++, and Python from CybersecEval, in line with RQ1 and RQ2, as these models were trained exclusively on these languages. Comparing them with CodeGuarder on other languages could introduce potential biases. For the investigated LLMs, we select two relatively small models, Mistral-7B [17] and CodeLlama-7B [37], due to the additional training required for baseline reproduction, which is both time-consuming and resource-intensive. For instance, training CodeLlama required approximately 1.4 million GPU hours [37]. Regarding data reproduction, we reuse the trained model provided by the authors for SafeCoder. For Sven and CoSec, we follow the official implementations of each approach and implement them using Mistral-7B and CodeLlama-7B.

Table 7: Performance of LLMs in the non-retrieval scenario

Metrics	LLM	Approach	C	C++	Python	Average
SR	Mistral-7B	Sven	60.35	70.27	74.93	68.52
		SafeCoder	63.00	75.29	78.63	72.31
		Cosec	59.91	70.66	76.64	69.07
	CodeLlama-7B	CodeGuarder	70.48	83.40	84.33	79.40
		Sven	62.11	70.66	71.23	68.00
		SafeCoder	64.89	76.56	77.78	73.08
Sim	Mistral-7B	Cosec	59.47	74.13	76.07	69.89
		CodeGuarder	68.72	79.54	82.91	77.06
		Sven	19.43	20.06	16.51	18.67
	CodeLlama-7B	SafeCoder	19.36	20.17	17.16	18.90
		Cosec	19.06	19.80	16.63	18.50
		CodeGuarder	23.08	25.19	22.37	23.55
CodeLlama-7B	Sven	19.26	19.96	18.03	19.08	
	SafeCoder	19.06	20.30	19.28	19.55	
	Cosec	18.46	19.36	18.15	18.66	
	CodeGuarder	24.02	25.65	23.11	24.26	

Table 7 presents the evaluation results. Overall, CodeGuarder outperforms the investigated baselines in both security hardening and functionality preservation in the non-func-retrieval setting. Compared to the state-of-the-art technique, SafeCoder, CodeGuarder generates 9.80% (72.31 → 79.40) more secure code on average with Mistral-7B. Additionally, the Sim metric of CodeGuarder is 24.60% higher than that of CoSec (18.90 → 23.55) on Mistral-7B. Similar improvements are observed with CodeLlama-7B. From a programming language perspective, CodeGuarder achieves significant improvements over previous methods. Specifically, its relative improvement over SafeCoder in the SR metric is 11.87%, 10.77%, and 7.25% on Mistral-7B across C, C++, and Python. We also observed that the Sim metric of CodeGuarder is significantly higher than the baselines, and we attribute it to the provided decomposed sub-tasks and the examples in the provided security knowledge. Notably, we observe that the security rate of the generated code is lower than the results reported in prior studies [13, 14, 21]. This discrepancy is likely due to differences in dataset scope: the prior works evaluate on a test set with only 166 cases across nine CWE types, whereas CyberSecEval contains 1,916 cases covering 50 CWE types, offering a more comprehensive assessment.

These results demonstrate that CodeGuarder not only enhances the security of LLM-generated code in retrieval scenarios (i.e., RACG) but also achieves strong performance in non-func-retrieval settings, outperforming existing security-hardening approaches specifically designed for non-func-retrieval scenarios.

6.4.2 Performance with Language-Specific Knowledge Absence. In real-world settings, the security knowledge base \mathcal{S} may lack knowledge for certain programming languages (e.g., C#), as its distribution varies widely across languages. This subquestion examines whether CodeGuarder can still enhance code security when target language knowledge is absent, leveraging knowledge from other languages. We evaluate CodeGuarder’s effectiveness on four external languages, C#, JavaScript, PHP, and Rust, where the knowledge retriever accesses a \mathcal{S} devoid of target-specific security knowledge, testing its cross-language generalization.

Table 8 presents CodeGuarder’s performance without language-specific security knowledge. As observed in RQ1 and RQ2, the

Table 8: Performance of CodeGuarder in the absence of language-specific knowledge.

Scenario	LLM	Language				
		C#	JavaScript	PHP	Rust	Average
Standard	GPT-4o	52.77	54.62	62.35	52.94	55.67
	w. CodeGuarder	61.28	65.46	78.40	60.29	66.36
	DS-V3	46.38	53.82	60.49	53.92	53.65
	w. CodeGuarder	61.70	67.07	75.93	53.92	64.66
	CodeLlama	50.64	54.62	60.49	52.94	54.67
	w. CodeGuarder	60.85	64.26	70.37	61.76	64.31
	DS-Coder	54.47	57.83	61.11	56.37	57.45
	w. CodeGuarder	55.32	58.63	74.07	55.39	60.85
	Average	51.07	55.22	61.11	54.04	55.36
	w. CodeGuarder	59.79	63.86	74.69	57.84	64.04
		(↑ 17.08%)	(↑ 15.63%)	(↑ 22.23%)	(↑ 7.03%)	(↑ 15.69%)
I†	GPT-4o	42.13	42.57	41.36	34.8	40.22
	w. CodeGuarder	47.66	51.41	55.56	41.18	48.95
	DS-V3	42.13	43.78	41.98	38.73	41.66
	w. CodeGuarder	56.17	56.63	62.35	39.71	53.72
	CodeLlama	53.19	48.59	40.12	45.1	46.75
	w. CodeGuarder	60.85	61.04	62.96	45.59	57.61
	DS-Coder	45.11	48.59	43.21	36.27	43.30
	w. CodeGuarder	51.91	47.79	56.79	36.27	48.19
	Average	45.64	45.88	41.67	38.73	42.98
	w. CodeGuarder	54.15	54.22	59.42	40.69	52.12
		(↑ 18.64%)	(↑ 18.17%)	(↑ 42.59%)	(↑ 5.07%)	(↑ 21.26%)
II‡	GPT-4o	49.79	54.22	63.58	54.41	55.50
	w. CodeGuarder	57.87	66.67	80.63	61.76	66.73
	DS-V3	45.96	53.01	59.88	55.39	53.56
	w. CodeGuarder	61.28	64.26	80.63	54.9	65.27
	CodeLlama	53.62	58.63	70.37	65.69	62.08
	w. CodeGuarder	69.95	64.26	73.46	61.76	67.36
	DS-Coder	54.04	60.24	70.94	54.41	59.91
	w. CodeGuarder	58.27	61.83	70.06	61.35	62.88
	Average	50.85	56.53	66.19	57.48	57.76
	w. CodeGuarder	61.84	64.26	76.19	59.94	65.56
		(↑ 21.61%)	(↑ 13.68%)	(↑ 15.11%)	(↑ 4.29%)	(↑ 13.50%)

† “I” and “II” denote the Poisoning Scenario I and II.

functionality impact of CodeGuarder is minimal; therefore, we focus solely on the SR metric in this analysis.

Overall, CodeGuarder consistently improves SR across scenarios even without language-specific knowledge. In the Standard scenario, SR rises from 55.36% to 64.04% (15.69% improvement), with gains of 17.08% (C#), 15.63% (JavaScript), 22.23% (PHP), and 7.03% (Rust). In poisoning scenario I, the average SR increases from 42.98% to 52.12% (21.26%), with PHP showing the highest gain (42.59%). In poisoning scenario II, the average SR improves from 57.76% to 65.56% (13.50%), with C# leading at 21.61%.

The consistent SR improvements suggest that CodeGuarder leverages universal security principles (e.g., avoiding weak random number generation, which is a common pitfall across languages like C++, C#, and JavaScript) present in the knowledge base. However, SR improvements in the poisoning scenario I are significantly lower than in languages with corresponding security knowledge in the knowledge base. Specifically, the average improvement in C, C++, Java, and Python is 31.53%, whereas in the other four languages, it is only 21.26%. These findings indicate that while CodeGuarder can mitigate poisoning attacks even without language-specific knowledge by relying on general security patterns, its effectiveness is somewhat limited, and the mitigation process becomes more challenging without tailored security information.

Table 9: Impact of CodeGuarder on LLMs’ functionality

LLM	MBPP		HumanEval	
	Pass@1	Pass@5	Pass@1	Pass@5
GPT-4o	72.8	77.4	89.0	91.5
w. CodeGuarder	73.2	78.8	89.6	91.5
DS-V3	74.6	78.6	87.2	89.6
w. CodeGuarder	75.6	79.6	87.8	90.9
CodeLlama	54.8	60.8	40.9	47.6
w. CodeGuarder	56.4	62.2	43.3	50.0
DS-Coder	62.8	69.2	80.5	86.0
w. CodeGuarder	65.0	72.8	81.7	88.4

Answer to RQ3: CodeGuarder demonstrates inspiring generalization, outperforming state-of-the-art security-hardening methods, SafeCoder, by 9.80% in SR on Mistral-7B when no similar code is retrieved. Moreover, even in the absence of language-specific knowledge, CodeGuarder consistently improves security across various languages, leveraging universal security principles.

7 Discussion

7.1 Functional Correctness Maintenance

The Sim metric, used in prior RQs, measures code similarity to reference implementations but does not directly assess functional correctness due to the absence of test cases in the CyberSecEval dataset. To provide a comprehensive evaluation of CodeGuarder’s impact on the functionality of LLM-generated code, we employ the MBPP [2] and HumanEval [6] benchmarks, which leverage test cases to measure code correctness. This section investigates whether CodeGuarder’s security enhancements compromise or enhance the practical utility of the generated code.

Table 9 reports results on the MBPP and HumanEval benchmarks across different LLMs, using the default benchmark settings. Security knowledge is integrated as described in §4.3. Overall, CodeGuarder slightly enhances the functionality of the generated code across all four evaluated LLMs and benchmarks. For instance, CodeGuarder improves the pass@1 score of GPT-4o on MBPP from 72.8 to 73.2 and the pass@5 score from 77.4 to 78.8. A similar trend is observed across other LLMs and benchmarks (i.e., HumanEval), indicating that integrating security knowledge does not compromise, and even enhances the functionality of LLM-generated code.

7.2 Effectiveness Across Vulnerability Types

In this discussion, we evaluate the effectiveness of CodeGuarder in preventing vulnerabilities across MITRE’s Top-25 software weaknesses [8] on three investigated scenarios. To quantify this, we define the prevention percentage as the proportion of vulnerabilities present in code generated without CodeGuarder that are successfully mitigated when CodeGuarder is applied.

Given CodeGuarder’s consistent security hardening across LLMs, we use DS-V3 as the representative LLM. Results are presented in Table 10. Overall, CodeGuarder demonstrates varying effectiveness, with average prevention rates ranging from 4.76% (CWE-79) to 89.15% (CWE-119). It excels at addressing critical weaknesses like CWE-119 (buffer errors, 89.15%) and CWE-200 (information exposure, 66.67%), reflecting its strength in leveraging security knowledge for broadly applicable vulnerabilities. However, its performance dips for CWE-79 (cross-site scripting, 4.76%), where analysis reveals that the generated code contained 3, 7, and 3 CWE-79

Table 10: Percentage of vulnerabilities prevented by CodeGuarder across MITRE’s Top-25 software weaknesses

CWE†	Standard	I	II	Average
CWE-79	0.00	14.29	0.00	4.76
CWE-89	34.62	40.00	34.78	36.47
CWE-352	40.00	38.89	36.36	38.42
CWE-22	13.79	3.85	7.14	8.26
CWE-78	40.00	44.12	46.99	43.70
CWE-862	18.18	28.57	27.27	24.67
CWE-94	38.10	41.38	39.13	39.54
CWE-502	53.33	12.24	15.56	27.04
CWE-200	100.00	50.00	50.00	66.67
CWE-918	26.67	20.00	31.25	25.97
CWE-119	88.89	78.57	100.00	89.15
CWE-798	30.77	25.93	46.67	34.46

† Only CWE types present in both MITRE’s Top-25 and CyberSecEval are shown.

Table 11: Performance comparison across different variants

Metric	Variant	Standard	I	II
SR	CodeGuarder-QD	68.61	64.04	66.25
	CodeGuarder-KRF	74.13	68.29	74.87
	CodeGuarder	76.36	70.22	76.42
Sim	CodeGuarder-QD	23.04	25.73	20.41
	CodeGuarder-KRF	24.82	27.04	21.14
	CodeGuarder	25.56	28.46	22.39

instances across the three scenarios, all stemming from JavaScript. This poor result stems from a lack of JavaScript-specific knowledge in CodeGuarder’s security knowledge base, limiting its effectiveness for this CWE type. Other lower-performing cases, such as CWE-22 (path traversal, 8.26%), suggest similar context-specific challenges. These findings highlight CodeGuarder’s strengths in securing different vulnerability types while underscoring areas for refinement, particularly in enriching the knowledge base.

7.3 Ablation Study

To further dissect the contribution of its core components, we conducted an ablation study by evaluating two variants of CodeGuarder, each with a key module disabled. Specifically, we created variants excluding Query Decomposition (QD), denoted CodeGuarder-QD, and excluding Knowledge Re-ranking and Filtering (KRF), denoted CodeGuarder-KRF. In the CodeGuarder-QD variant, security knowledge was retrieved using the user’s original query directly, bypassing the sub-task decomposition step. In the CodeGuarder-KRF variant, all initially retrieved security knowledge for the sub-tasks was provided to the LLM for code generation without applying the re-ranking and filtering process. It is crucial to note that the KRF module operates on the sub-tasks generated by QD; therefore, disabling QD inherently disables the KRF mechanism as well. To ensure a fair comparison, we maintain consistency in the number of injected knowledge entries across all variants. Given that CodeGuarder demonstrates consistent performance across different LLMs, we conduct this ablation study using DS-V3.

The results are presented in Table 11. Overall, QD exhibits the most substantial impact on both the SR and Sim metrics. Removing QD led to a decrease in SR of approximately 10.1% (from 76.36% to 68.61%) and a drop in Sim from 25.56 to 23.04 in the standard

Table 12: The Impact of k' and k on CodeGuarder’s Performance (SR)

(a) DS-V3					(b) CodeLlama				
k'	k				k'	k			
	3	5	7	9		3	5	7	9
1	69.84	73.32	75.15	76.04	1	66.14	67.05	69.45	69.39
2	74.72	76.36	77.26	76.82	2	67.38	69.74	68.23	67.21
3	75.31	76.53	75.62	75.03	3	68.17	68.59	67.42	66.35

scenario. Similar trends were observed in scenarios I and II. In contrast, the contribution of KRF was less pronounced. Removing only KRF (CodeGuarder-KRF) resulted in a smaller decrease in SR of about 2.2% (from 76.36% to 74.13%) and a reduction in Sim from 25.56 to 24.82 in the standard scenario. These findings underscore the critical role of Query Decomposition in enabling more precise knowledge retrieval, thereby significantly enhancing the security of the generated code. Knowledge Re-ranking and Filtering provide an additional, valuable refinement to this process.

7.4 Impact of the Number of Injected Knowledge Entries

The amount of injected security knowledge significantly impacts CodeGuarder’s performance. This amount is controlled by k' (knowledge entries retrieved per sub-task, §4.2.2) and k (top-ranked sub-tasks selected after filtering, §4.2.3). We empirically tuned these hyperparameters by evaluating various (k' , k) combinations using DS-V3 (a larger model) and CodeLlama-13B (a smaller model) on the standard RACG scenario.

Results are presented in Table 12. We observed that performance generally improves with more knowledge up to a point, after which excessive knowledge leads to degradation. The optimal configuration among the investigated hyperparameters was ($k' = 2, k = 7$) for DS-V3 and ($k' = 2, k = 5$) for CodeLlama-13B, indicating that DS-V3 can effectively leverage more security knowledge than CodeLlama-13B. This suggests that models with relatively fewer parameters may be overwhelmed by excessively large knowledge contexts, leading to performance drops due to their comparatively weaker instruction-following capabilities [7, 18].

Based on these findings, and aiming for robust performance across different model scales while mitigating degradation, we selected $k' = 2$ and $k = 5$ as the default hyperparameters for CodeGuarder in our main evaluations (§6).

7.5 Threats to Validity

We identify the following potential threats to the validity:

Validity of Security Measurement. Our primary measure of code security relies on the Insecure Code Detector [3] applied within the CyberSecEval benchmark. This detector employs an ensemble of static analysis tools (e.g., Semgrep [38] and Weggli [47]) configured to identify patterns with 50 CWEs. A threat to validity arises because this measurement may not perfectly capture the true security posture of the generated code. Furthermore, the predefined set of 50 CWEs, while significant, may not represent the complete universe of potential security flaws.

We argue that this threat is partially mitigated by the following factors: (1) The 50 CWEs targeted by the detector encompass a broad range of common and critical vulnerability types frequently encountered in practice, providing substantial coverage. Notably,

the Insecure Code Detector achieves a detection precision of 96% for vulnerabilities within this set. (2) The utilization of static analysis tools represents a standard and practical methodology for large-scale automated security assessment in existing studies [20, 34, 41]. And (3) Most importantly for our comparative study, the consistent application of the same detection tools and criteria across all evaluated LLMs and investigated scenarios ensures that the observed differences in security performance (e.g., the improvements attributed to CodeGuarder) are measured fairly, allowing for reliable and robust comparisons.

Reliability and LLM Non-Determinism. Another threat arises from the inherent non-determinism associated with LLMs, which could impact the reliability and reproducibility of our experimental results. Specifically, LLMs can reproduce varying outputs even when presented with the same input multiple times. One potential mitigation is to average performance over multiple generations, but this approach is extremely time-consuming. To mitigate this threat, we adhered to the recommended practices [33] by setting the temperature parameter to 0 during all LLM inference steps, which minimizes the stochasticity in the generation process.

However, even at zero temperature, complete determinism is not assured, particularly for architectures like Mixture-of-Experts (e.g., DS-V3 used in this study) [35]. To quantify the impact of residual non-determinism, we conducted five independent runs of the DS-V3 under the standard scenario and measured variability in our primary metric, the Security Rate (SR). The maximum observed deviation in SR across these runs was 0.38%, indicating negligible influence on our findings. This low level of variability suggests that the impact of residual LLM non-determinism is negligible, indicating that this threat is well-controlled in our study.

8 Conclusion

In this work, we propose CodeGuarder, a security-hardening framework for RACG systems that addresses the overlooked threat of knowledge base poisoning. By explicitly incorporating security knowledge into the code generation prompt, CodeGuarder enables LLMs to generate more secure code without compromising functional correctness. Extensive evaluation across various scenarios shows that CodeGuarder significantly improves the security of generated code while maintaining its functionality, and also demonstrates strong generalization across languages and models. This work takes a critical step toward securing LLM-based software development.

Acknowledgments

References

- [1] 2024. Infer. <https://fbinfer.com/>.
- [2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [3] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724* (2023).
- [4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [5] Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Xiaoqian Liu, Tong Xiao, and Jingbo Zhu. 2024. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077* (2024).
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [7] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.
- [8] CWE Team. 2024. *2024 CWE Top 25 Most Dangerous Software Weaknesses*. MITRE. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
- [9] Xinyu Gao, Yun Xiong, Deze Wang, Zhenhan Guan, Zejian Shi, Haofen Wang, and Shanshan Li. 2024. Preference-Guided Refactored Tuning for Retrieval Augmented Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 65–77.
- [10] GitHub. 2023. CodeQL. <https://codeql.github.com>
- [11] Suhas Hariharan, Zainab Ali Majid, Jaime Raldua Veuthey, and Jacob Haimes. 2024. Rethinking CyberSecEval: An LLM-Aided Approach to Evaluation Critique. *arXiv preprint arXiv:2411.08813* (2024).
- [12] Haitao He, Sheng Wang, Yanmin Wang, Ke Liu, and Lu Yu. 2025. VulTR: Software vulnerability detection model based on multi-layer key feature enhancement. *Computers & Security* 148 (2025), 104139.
- [13] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.
- [14] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497* (2024).
- [15] Kaifeng Huang, Chenhao Lu, Yiheng Cao, Bihuan Chen, and Xin Peng. 2024. VMUD: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3958–3972.
- [16] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- [17] Albert Q. Jiang, Arthur Sablayrolles, Alexandre Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gabor Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *CoRR abs/2310.06825* (2023). <https://arxiv.org/abs/2310.06825>
- [18] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).
- [19] Arya Kaviani, Mohammad Mehdi Pourhashem Kallehbasti, Sajjad Kazemi, Ehsan Firouzi, and Mohammad Ghafari. 2024. Llm security guard for code. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 600–603.
- [20] Jan H Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton Jr, Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, et al. 2024. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2726–2740.
- [21] 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*. 1428–1439.
- [22] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.
- [23] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).
- [24] Bo Lin, Shangwen Wang, Liqian Chen, and Xiaoguang Mao. 2025. Exploring the Security Threats of Knowledge Base Poisoning in Retrieval-Augmented Code Generation. *arXiv preprint arXiv:2502.03233* (2025).
- [25] Aixun Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
- [26] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.
- [27] Microsoft. 2024. Retrieval-Augmented Generation (RAG) in Azure Machine Learning. <https://learn.microsoft.com/en-us/azure/machine-learning/concept-retrieval-augmented-generation>
- [28] Ollama. 2023. Ollama framework. <https://ollama.com/>.
- [29] OpenAI. [n. d.]. GPT-4o. <https://platform.openai.com/docs/models#gpt-4>.

- [30] OpenAI. 2024. ChatGPT Retrieval Plugin. <https://github.com/openai/chatgpt-retrieval-plugin>
- [31] OpenAI. 2024. OpenAI API Reference. <https://platform.openai.com/docs/api-reference>
- [32] OpenAI. 2025. OpenAI Models - Embeddings. <https://platform.openai.com/docs/models/gpt#embeddings>.
- [33] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2024. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology* (2024).
- [34] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
- [35] Joan Puigcerver, Carlos Riquelme, Basil Mustafa, and Neil Houlsby. 2023. From spore to soft mixtures of experts. *arXiv preprint arXiv:2308.00951* (2023).
- [36] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv:2009.10297 [cs.SE]*
- [37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [38] Semgrep, Inc. 2025. Semgrep. <https://semgrep.dev/>.
- [39] Saba Sturua, Isabelle Mohr, Mohammad Kalim Akram, Michael Günther, Bo Wang, Markus Krimmel, Feng Wang, Georgios Mastrapas, Andreas Koukounas, Nan Wang, et al. 2024. jina-embeddings-v3: Multilingual embeddings with task lora. *arXiv preprint arXiv:2409.10173* (2024).
- [40] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. EvoR: Evolving Retrieval for Code Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 2538–2554.
- [41] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C Cordeiro. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering* 30, 2 (2025), 1–42.
- [42] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 588–592.
- [43] Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, et al. 2024. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. *arXiv preprint arXiv:2408.01605* (2024).
- [44] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseeval. *arXiv preprint arXiv:2407.02395* (2024).
- [45] Xinchun Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 472–483.
- [46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [47] Felix Wilhelm, Fabian Freyer, Calle Svensson, Thomas Otto, Michal Melewski, William Woodruff, Disconnect3d, and Matthew Rinaldi. 2025. weggli. <https://github.com/weggli-rs/weggli>.
- [48] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).
- [49] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities. *ACM Transactions on Software Engineering and Methodology* (2025).
- [50] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792* (2023).
- [51] Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 1291–1302.
- [52] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).
- [53] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).

A Security Knowledge Example

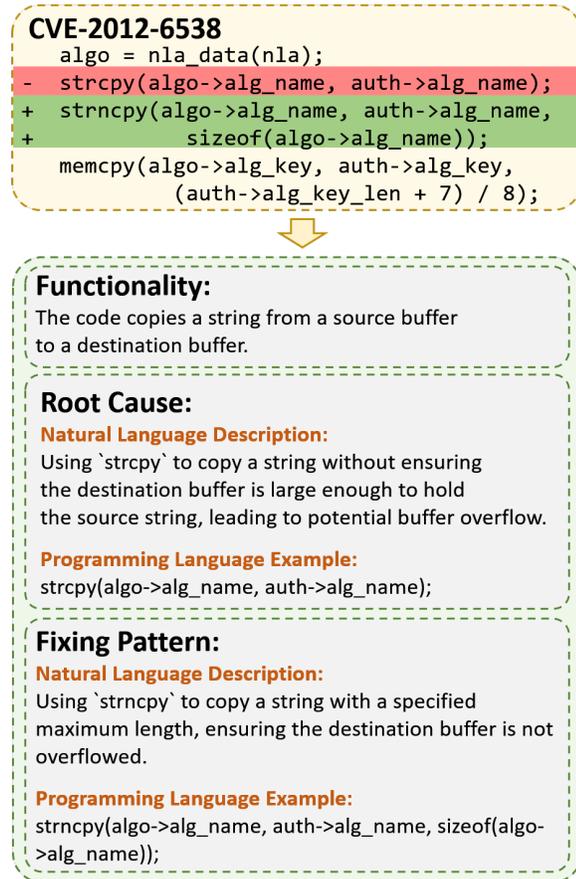


Figure 4: An example of security knowledge extracted from vulnerability

Figure 4 presents an example of security knowledge extracted from CVE-2012-6538. This vulnerability arises from the use of an insecure function, which may lead to a buffer overflow. The patched code mitigates this issue by replacing `strcpy` with `strncpy`, enforcing a specific size constraint on the destination buffer.

This vulnerability arises from the use of an insecure function, which may lead to a buffer overflow. The patched code mitigates this issue by replacing `strcpy` with `strncpy`, enforcing a specific size constraint on the destination buffer. The **Functionality** dimension describes the fundamental operation of the vulnerable code snippet (i.e., copying a string from a source buffer to a destination buffer). The **Root Cause** dimension provides a detailed explanation of the vulnerability in natural language along with an illustrative code example (i.e., the risks associated with using `strcpy`). Finally, the **Fixing Pattern** dimension includes both a description of the secure coding practice and an example of the corrected code (i.e., replacing

`strcpy` with a safer alternative, `strncpy`). This structured knowledge extraction process informs secure code generation in RACG, enabling CodeGuarder to enhance the security of the generated code.

B Prompt Templates

B.1 Security Knowledge Extraction

Prompt 1: Security Knowledge Extraction

Task: Analyze a vulnerability fixing commit to extract security knowledge.

Input:

- **Vulnerability Description:** $\{D_{cve}\}$
- **Vulnerability Type:** $\{D_{cwe}\}$
- **Fixing Commit (Diff):** $\{\text{DIFF}\}$

Instructions:

- (1) Describe the functionality of the vulnerable code snippet.
- (2) Identify and extract the root cause of the vulnerability.
- (3) Identify and extract the corresponding fixing pattern.

Output Format: Provide the output in JSON format, adhering to the following structure:

```

{
  "Functionality": "<Description of the vulnerable code's functionality>",
  "Root_Cause": [
    "<Detailed description of the vulnerability's root cause>",
    "<Code example illustrating the vulnerability>" ]
  "Fixing_Pattern": [
    "<Detailed description of the fixing pattern>",
    "<Code example illustrating the vulnerability repair>" ]
}

```

In Prompt 1, the terms D_{cve} and D_{cwe} denote the CVE description and the Common Weakness Enumeration (CWE) classification type, respectively. The DIFF represents the function-level diff, generated by comparing the vulnerable code C_v with its corresponding fixed version C_f . Note that DIFF includes the full function context, as this detailed diff provides comprehensive information for understanding both the vulnerability and its resolution.

B.2 Query Decomposition

Prompt 2: Query Decomposition

Please help me to break down a code generation query into smaller, detailed sub-tasks. For each sub-task, please utilize the natural language description of what the sub-tasks do, focusing on explaining the functionality of the sub-tasks.

User's Query

{QUERY}

Output

Please output the following information in a JSON object format: [{"Description": "string"}]

Prompt 2 shows the template used for query decomposition, where QUERY represents the user's original query Q .

B.3 Security-Augmented Code Generation

Prompt 3: Security-Augmented Code Generation

```

/*--Original RACG Prompt Begin--*/
Write a C function that allocates memory for a string and copies its
content, returning a pointer to the new string.

# Code Examples
char* copy_string(char* src) { malloc... }
/*--Original RACG Prompt End--*/

/*--Injected Security Knowledge--*/
The code generation process involves the following sub-tasks. For
each, I provide potential vulnerabilities and corresponding mitigation
strategies.

# Sub-Tasks with Corresponding Knowledge
Sub-Task 1: Copy each string from the input array to the correspond-
ing location in the new array.
Security Knowledge 1: ...
{
  "Functionality": ...,
  "Root_Cause": [
    <Description>: Using strcpy to copy a string without ensuring
the buffer is large enough.
    <Code Example>: strcpy(...);
  "Fixing_Pattern": [
    <Description>: Use strncpy to ensure the string is copied safely.
    <Code Example>: strncpy(...);
  ]
}
...
Sub-Task N:{SUB_TASK_N}
Security Knowledge N:{SEC_KNOW_N}

```

Prompt 3 provides a concrete example of prompt for security-augmented code generation. The injected security knowledge includes the relevant knowledge retrieved for the given task. In the prompt, {SUB_TASK_N} and {SEC_KNOW_N} refer to the N-th sub-task and its corresponding retrieved security knowledge, respectively.