Fortifying LLM-Based Code Generation with Graph-Based Reasoning on Secure Coding Practices

Rupam Patir, Keyan Guo, Haipeng Cai, Hongxin Hu
University at Buffalo
{rupampat, keyanguo, haipengc, hongxinh}@buffalo.edu

Abstract—The code generation capabilities of Large Language Models (LLMs) have transformed the field of software development. However, this advancement also presents significant security challenges, as LLM-generated code often contains vulnerabilities. One direction of research strengthens LLMs by injecting or refining security knowledge through curated datasets, model tuning, or static analyzers. While effective in certain settings, these methods can be resource-intensive, less adaptable to zero-day vulnerabilities, and often inapplicable to proprietary models. To address these challenges, we introduce GRASP, which explores a new direction that focuses on structured reasoning over Secure Coding Practices (SCPs) rather than additional training or external feedback. GRASP comprises two key ideas: (1) an SCP graph that organizes SCPs into a Directed Acyclic Graph (DAG) capturing dependencies and relationships, and (2) a graph-based reasoning process that systematically guides LLMs through relevant SCPs for code generation. This design enables interpretable, model-agnostic, and scalable security improvements, particularly for previously unseen vulnerabilities. Our evaluation shows that GRASP consistently achieves Security Rates (SR) exceeding 80% across multiple LLMs, and delivers up to 88% improvements over baselines on zero-day vulnerabilities.

1. Introduction

The rapid advancements in Large Language Models (LLMs) have transformed software development by enhancing coding capabilities [1], [2], [3], [4], [5], [6]. Research highlights the productivity benefits of these models in software development, with GitHub Copilot users completing tasks 55% faster, experiencing improved job satisfaction, and reduced cognitive load [7], [8]. Google's findings further highlight the advantages of ML-enhanced code completion, demonstrating a 6% reduction in time between builds and tests with single-line ML completion, along with a 25-34% user acceptance rate for code suggestions [9].

Despite these advancements, critical security concerns persist in adopting LLMs for code generation. Studies have shown that LLM-generated code frequently contains vulnerabilities. For instance, GitHub Copilot was found to produce vulnerable code in approximately 40% of cases [10], [11]. More recently, Mou et al. [12] reported that popular LLMs such as GPT-4o generate secure code only 70–75% of the time. These rates are untenable in practice, as insecure code can readily lead to data breaches, privilege escalation, or widespread system compromise.

To mitigate these concerns, researchers have proposed several techniques to strengthen LLMs for secure code generation. SVEN [13] applied prefix tuning to improve generation security. SafeCoder [14] extended this approach using instruction tuning to provide more flexible control over secure code generation. PromSec [15] introduced prompt-level optimization through graph-based adversarial training, leveraging curated vulnerability datasets and external static analyzers such as Bandit [16] and SpotBugs [17] to guide the refinement process. Together, these methods represent one direction of research that strengthens LLMs by equipping them with additional security knowledge or refining it through external signals. While effective, this direction faces persistent challenges, including reliance on curated datasets that miss zero-day vulnerabilities, the need for model access unavailable in proprietary systems, added overhead from external analyzers and limited interpretability.

In this paper, we pursue a novel direction that focuses on ensuring the reliable and systematic operationalization of security knowledge. Recent work shows that LLMs already exhibit awareness of security concepts relevant to software development [18], [19]. In this case, the challenge is not acquiring knowledge but ensuring its consistent application during code generation. This difficulty mirrors a longstanding issue in human software development and reflects the well-documented gap between *security theory* and *coding practice* [20]. Developers may understand secure design concepts abstractly but fail to apply them under real-world conditions. To address these gaps, the software engineering community has long relied on Secure Coding Practices (SCPs) [21], [22], [23], which provide the structured discipline needed to translate abstract knowledge into consistent implementation. Empirical studies [20] further confirm that adherence to SCPs enabled developers to produce more secure code in practice, motivating us to consider whether similar structured practices can help LLMs.

We therefore extend this precedent by adapting it to LLMs. Just as disciplined methods helped developers reliably apply what they already knew, structured mechanisms can guide LLMs in transforming their latent security knowledge into secure outputs. GRASP builds on this by operationalizing SCPs for LLM-based code generation. It does

so through two key ideas: (i) an SCP Graph that organizes practices into a Directed Acyclic Graph (DAG) encoding their dependencies and relationships, and (ii) a graph-based reasoning process that systematically applies these practices in context. In this way, GRASP represents a novel direction relative to prior work, shifting the focus from augmenting security knowledge through retraining or curated datasets to ensuring its reliable and systematic operationalization. This design yields natural benefits such as generalization to zero-day vulnerabilities, model agnosticism, freedom from external analyzers, and improved interpretability.

The key contributions of this paper are as follows:

- **We investigate the role of Secure Coding Practices (SCPs) in conditioning LLMs for secure code generation.** SCPs, widely used by developers to prevent vulnerabilities, can also be leveraged to guide LLMs toward more secure outputs. We advance this premise by modeling SCPs as a structured graph and analyzing their effectiveness for security-aware generation.
- **We propose GRASP[1], a reasoning-driven framework that leverages an SCP Graph to fortify the code generation process.** The SCP Graph is a Directed Acyclic Graph that encodes dependency relationships among SCPs. GRASP traverses this graph dynamically based on task relevance, incrementally applying security transformations while maintaining logical consistency and functional correctness.
- **We construct a benchmark dataset for secure code generation, annotated with unit tests to enable joint evaluation of security and functionality.** The benchmark consists of a diverse collection of CWE-based scenarios, each paired with unit tests to ensure functional correctness. To the best of our knowledge, this is the first benchmark for secure code generation that simultaneously accomplishes **both** (a) the use of natural language prompts instead of code completion prompts, and (b) the joint evaluation of security and functional correctness.
- **We conduct comprehensive experiments across multiple LLMs, CWEs, and zero-day CVEs.** GRASP improves the overall Security Rate (SR) to over **80%** for Claude, GPT-4o, Gemini and Llama3, while maintaining functional correctness. It also generalizes to unseen vulnerabilities, achieving SR gains of up to **88%** over baseline methods on real-world CVEs.

2. Background and Related Work

In this section, we provide the necessary background knowledge and discussion of closely related work.

2.1. LLMs in Code Generation

Large Language Models (LLMs) have significantly impacted the field of code generation, offering new possibilities for automating and assisting in software development tasks.

---

1. We will make our data and code publicly available in advance of the paper's publication.

Brown et al. [24] demonstrated the potential of large-scale language models to perform various tasks, including code generation, with minimal task-specific fine-tuning. Chen et al. [25] further explored the capabilities of LLMs trained specifically in code, highlighting their potential and limitations. The development of CodeBERT by Feng et al. [26] showcased the effectiveness of pre-training models on both programming languages and natural language descriptions.

LLMs have emerged as prominent tools for software developers in code-generation tasks. Proprietary models such as OpenAI's GPT series [1], Anthropic's Claude [2], and Google's Gemini [3] offer advanced code generation features with strong instruction-following abilities. They are designed to understand and generate code across various programming languages. GitHub Copilot [7], built on OpenAI's Codex [27], is specifically tailored for code completion and generation tasks within development environments. In contrast, open-source models like Meta's LLama3 [28] and Salesforce's CodeGen [29] provide accessible alternatives. These models differ in their accessibility, initial training focus, and level of instruction tuning, with proprietary models often offering more advanced performance for code-related tasks [30].

2.2. Vulnerable Code

Vulnerable code encompasses software flaws that can be exploited to compromise the security of systems, potentially leading to significant issues such as unauthorized access and data breaches. Addressing these flaws is crucial for ensuring the security and integrity of software applications [31].

The Common Weakness Enumeration (CWE) framework [32], maintained by the MITRE Corporation, categorizes these vulnerabilities into a structured list of weaknesses that can lead to security risks. As part of their efforts, MITRE maintains a list of the Top 25 CWEs, which highlights the most critical and prevalent weaknesses in software [33]. For example, SQL Injection (CWE-89) is a prevalent vulnerability that allows attackers to manipulate SQL queries through malicious input, potentially exposing or altering database content. Cross-site scripting (XSS) (CWE-79) is a vulnerability that allows attackers to inject malicious scripts into web applications, potentially leading to unauthorized access or data theft. Cross-Site Request Forgery (CSRF) (CWE-352) tricks users into performing unintended actions on a Web application. Insecure Deserialization (CWE-502) involves the unsafe handling of serialized data, which can be exploited to execute arbitrary code or alter application data.

The impact of these vulnerabilities can be substantial. Data breaches resulting from such flaws compromise user privacy, incur significant financial losses due to remediation efforts and downtime, and cause reputation damage that erodes customer trust [34], [35]. Effective detection and mitigation of these vulnerabilities involves techniques such as static code analysis [36], [37], [38] and dynamic code analysis [39], [40], [41], complemented by manual code reviews.

2.3. Secure Coding Practices

Secure coding practices (SCPs) are essential for coaching software developers in the development of resilient and secure software. They encompass a variety of techniques designed to prevent vulnerabilities and ensure that the code behaves securely under various conditions [42], [43]. For instance, to prevent SQL Injection, which occurs when malicious input manipulates SQL queries, it is crucial to use parameterized queries and prepared statements. This ensures that user input is treated as data rather than as executable code. Similarly, to mitigate cross-site scripting (XSS), it is vital to ensure that user input is properly sanitized before being rendered.

Resources such as OWASP Secure Coding Practices [21], CERT Secure Coding Standards [22], and Microsoft Secure Coding Guidelines [23] provide comprehensive guidance, including centralized input validation, secure data encoding/decoding, and robust authentication and session management. Following these practices helps developers significantly reduce the risk of vulnerabilities.

2.4. LLM-Based Code Generation Fortification

Improving the security of LLM-based code generation has gained increasing attention in recent research. He et al. [13] introduced SVEN, a control method that applies prefix-tuning to steer models toward generating secure or insecure code. Rather than updating the model's original weights, this approach trains only lightweight prefix parameters, thereby reducing the number of trainable components while retaining flexibility for code generation tasks. Building on this direction, He et al. [14] proposed SafeCoder, which performs security-focused instruction tuning using a large-scale dataset of verified vulnerability fixes gathered from GitHub commits. SafeCoder jointly optimizes for both security and functional utility, yielding a notable 30% improvement in code security across diverse tasks. However, such approaches require considerable computational resources and may overfit to specific vulnerability patterns. Beyond model tuning, prompt engineering has been leveraged to craft prompts that encourage secure coding [44].

Recent work by PromSec [15] advances this paradigm by introducing a generative adversarial graph neural network (gGAN)-based framework to iteratively optimize prompts for secure code generation. PromSec employs a dual-objective contrastive learning strategy to simultaneously address vulnerability mitigation and functional correctness, reducing reliance on iterative LLM inferences while achieving transferability across models and programming languages.

3. Motivation and Observation

3.1. Risk of LLM-based Code Generation

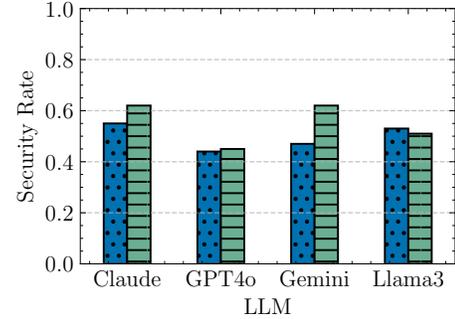The rapid adoption of LLMs in software development has spurred notable advancements in code generation [8],

Figure 1: SR of Base Model ▒ and Zero-Shot Method ▒.

[9]. Although LLMs have made significant strides in code generation, they still present substantial security risks [44], [45]. Studies have shown that AI tools for code generation like GitHub Copilot generate vulnerable code about 40% of the time [10], [11]. Khoury et al. found that, out of 21 code generation tasks, ChatGPT [1] produced secure code for only 5 tasks that met security standards [45].

In our study, we assessed the tendency of LLMs to generate vulnerable code. To do this, we evaluated OpenAI's GPT-4o-mini [46], Anthropic's Claude-3-haiku-20240307 [47], Google's Gemini-1.5-flash-latest [48] and Meta's Llama-3-8B-Instruct [49]. We created 36 natural language code generation prompts targeting 8 different CWEs, with 1–7 prompts per CWE, each designed to generate Python code that potentially contains specific vulnerabilities. We tested two prompting methods: (1) using the Base Model without any modifications, (2) using a Zero-Shot prompt that included an additional security-focused instruction: *"You are a Security Engineer and you develop code that adheres to secure coding practices."* We generated 25 samples for each prompt across all LLMs and prompt types, resulting in 7,200 samples of which we retained a total of 7,141 syntactically correct samples.

To detect vulnerabilities in the samples, we deployed CodeQL [36], a state-of-the-art static analysis tool. Each initial prompt was linked to a specific CWE and a corresponding CodeQL query, enabling the evaluation of the samples against relevant security standards. We use the security rate (SR), defined as the proportion of valid samples that are secure, as our evaluation metric, as detailed in Equation 1 in Section 5.2.

As shown in Figure 1, our results indicate that all four models consistently generate vulnerable code. Even when explicitly prompted to prioritize security, improvements in security rate were limited. For example, Claude's security rate increased from 0.55 (Base) to 0.62 (Zero-Shot), and Gemini's rose from 0.47 to 0.62. GPT-4o showed only a minor improvement, going from 0.44 to 0.45. Llama-3, on the other hand, saw a slight decline from 0.53 to 0.51. This can be attributed to its smaller model size and limited ability to reason about security-relevant patterns. These findings highlight ongoing security challenges in LLM code generation, even with security-focused prompting, underscoring

TABLE 1: Comparison of Secure Code Generation Approaches.

Property	SVEN	SafeCoder	PromSec	GRASP
Closed-source LLM compatible	○	○	●	●
No re-training or fine-tuning needed	○	○	◐	●
No vulnerability dataset required	○	○	○	●
No external feedback tools	●	●	○	●
Interpretable security reasoning	○	○	○	●

Note: ● indicates that the model has the property; ◐ indicates partial or conditional support; ○ indicates that it does not.

the need for structured methods to fortify LLM-based code generation.

3.2. Challenges with Approaches to Secure Code Generation

Recent work has introduced various techniques for the fortification of LLM-based code generation, ranging from fine-tuning strategies to prompt-level interventions. These approaches exemplify one direction of research, which strengthens LLMs by adding or refining security knowledge through training, curated data, or external analyzers. While these methods have led to measurable gains, they remain constrained by fundamental limitations that hinder broad applicability and long-term maintainability. We highlight four key challenges that motivate the design of GRASP.

Dependence on Curated Vulnerability Datasets: SVEN [13] and SafeCoder [14] harden LLMs through prefix-tuning and instruction tuning, respectively, with both methods requiring large, curated datasets of vulnerability-fix pairs. PromSec [15], while not tuning the model itself, also relies on curated vulnerability data to train a separate generative adversarial graph network (gGAN). Although effective on previously seen patterns, all three approaches are tightly coupled to specific vulnerability datasets and require ongoing retraining or reoptimization as security guidelines evolve. This dependence limits their ability to generalize to zero-day vulnerabilities and reduces scalability across new or changing domains. Zero-day vulnerabilities, previously unseen weaknesses, pose a severe challenge for security hardening systems. Because such vulnerabilities are absent from the curated vulnerability datasets used by most existing approaches, these systems often fail to detect or mitigate them. The evolving nature of zero-day vulnerabilities demands methods that can generalize security reasoning beyond memorized patterns.

Limited Accessibility across Models: Tuning-based techniques rely on access to model weights for prefix or instruction tuning, which is often infeasible in practice. SVEN [13] and SafeCoder [14], for instance, are incompatible with proprietary LLMs such as GPT-4o or Claude. This restriction severely limits deployment in many real-world settings where only black-box model access is available.

Dependence on External Feedback Mechanisms: PromSec [15] requires iterative prompt refinement using external static analyzers such as Bandit [16] or SpotBugs [17]. While this strategy can help detect certain vulnerabilities, it introduces non-trivial infrastructure dependencies and runtime overhead. Its performance also relies on the effectiveness of these external tools.

Lack of Interpretability: Existing approaches provide limited visibility into how or why specific security transformations are applied. Because their behavior is either learned through gradient-based updates or driven by opaque feedback loops, understanding the reasoning behind code modifications is difficult. This poses challenges for developers, auditors, and security engineers tasked with verifying correctness.

These challenges underscore the necessity of a new method that circumvents the limitations of prior approaches. In this regard, we introduce GRASP, a reasoning-centered framework for secure code generation. GRASP does not require access to model weights, curated vulnerability datasets, or external analysis tools, while offering an interpretable mechanism for fortifying the code generation process. By conditioning generation on SCPs rather than historical vulnerability corpora, GRASP is naturally equipped to generalize to previously unseen, zero-day vulnerabilities. Table 1 highlights the key distinctions between GRASP and existing methods.

4. GRASP Design

4.1. Design Intuition and Overview

Whereas the prior direction of work strengthens LLMs by adding or refining security knowledge through training, curated data, or analyzers, our complementary direction focuses on the systematic application of existing knowledge. SCPs offer a principled foundation for this direction for fortifying LLM code generation. Unlike curated vulnerability datasets or tuning-based approaches that focus on specific code instances, SCPs are generalized, human-readable principles widely adopted for guiding secure software development. Expressed in natural language, SCPs align well with the strengths of LLMs, allowing them to be readily understood and reasoned about without requiring task-specific tuning. GRASP builds on this foundation by introducing a reasoning-driven framework that systematically applies SCPs during code generation.

By leveraging SCPs rather than vulnerability-specific data, GRASP promotes generalizable security behavior that extends beyond known vulnerability patterns. This enables the framework to address a broad range of threats, including
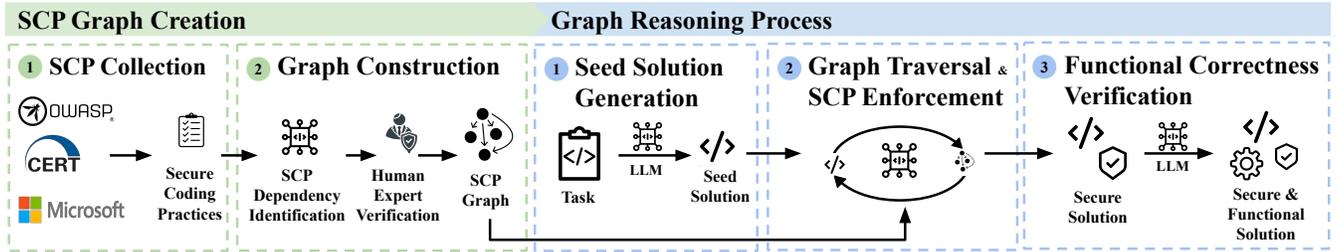
Figure 2: Overview of GRASP.

previously unseen zero-day vulnerabilities, without relying on curated datasets or model retraining.

Furthermore, GRASP eliminates the need for external static analyzers by relying on internal reasoning to determine which practices to apply. This removes the dependence on tool-specific integrations or runtime feedback, while still allowing the model to apply SCPs based on code context selectively. In addition, by organizing SCPs into a sequence of interpretable refinement steps, GRASP ensures transparency in security decision-making and allows easier auditing and understanding of the resulting output.

As illustrated in Figure 2, GRASP proceeds in two main phases. First, it constructs an SCP Graph by automatically extracting and organizing established guidelines in a dependency-aware manner, followed by final manual verification. Next, leveraging this graph, it produces an initial candidate solution and iteratively refines it via graph-guided reasoning over the relevant practices, ensuring that the resulting code is both secure and functionally correct.

4.2. SCP Graph

4.2.1. Structured Representation of SCPs. In pursuing this reasoning-oriented direction, SCPs offer a principled basis for fortifying LLM code generation, but applying them effectively introduces unique challenges. In practice, there are dozens of SCPs covering diverse concerns, from input validation to output encoding to error handling, and these practices are often interdependent. Some SCPs are only applicable in specific contexts, while some must be applied in a particular order to be effective. For example, SCPs for implementing error handling and logging should only be applied after enforcing input validation; otherwise, invalid input may propagate into logs or error messages, potentially exposing sensitive information or enabling injection attacks.

When SCPs are presented as an unordered list, LLMs may apply them in a redundant, irrelevant, or incorrectly sequenced manner, which can fail to improve the security of their generated code and may even introduce new vulnerabilities. Additionally, providing all SCPs at once can exceed the model's effective context window, leading to overload and reducing its ability to apply the practices accurately and effectively.

To address these challenges, we design the *SCP Graph*, a structured representation that organizes SCPs into a Directed

Acyclic Graph (DAG). Each node corresponds to an individual SCP, while edges encode semantic relationships such as ordering constraints and specificity hierarchies. This structure enables GRASP to reason systematically over SCPs, selecting only the practices relevant to the given context and applying them in a dependency-aware manner.

4.2.2. SCP Graph Design. Nodes in the graph represent the SCPs expressed in natural language. These nodes are designed to be self-contained, meaning each one encapsulates a specific security principle, such as "Validate all file paths before access" or "Use parameterized queries for database operations." At the same time, nodes are connected through edges that capture their dependency relationships. This structure allows each SCP to be evaluated and applied independently based on the context of the code, while still preserving its links to related practices for broader reasoning. Edges in the graph capture the relationships between SCP nodes, categorized into two types:

i. **Specificity Relationships** which indicate connections between general SCPs and their more specific implementations. For example, a parent node containing "Ensure robust security measures for database management" connects to more specific child nodes like "Use strongly typed, parameterized queries" and "Implement proper error handling for database operations". This hierarchical structure allows GRASP to move from broad security principles to concrete ones.

ii. **Sequential Relationships** which represent a required sequence in the implementation of SCPs. For example, an edge from "Implement input validation" to "Implement error handling for validation failure" indicates that proper error handling can only be implemented after input validation is in place. These relationships ensure that SCPs are applied in a logical and effective order.

This design organizes SCPs hierarchically from general to specific, supports multiple implementation paths for flexibility, and structures branches so that irrelevant subtrees can be bypassed. Furthermore, this organization improves the clarity and maintainability of dependencies and relationships.

4.2.3. SCP Graph Construction. SCPs can be drawn from several authoritative sources, including OWASP [21], CERT [22], and Microsoft's secure coding standards [23]. While our methodology is general and supports integrating

practices from any such reference, in this work we focus exclusively on the OWASP Secure Coding Practices Checklist, selected for its breadth, accessibility, and widespread adoption within the developer community.

From OWASP's checklist, we manually filter practices using two criteria to ensure suitability for code generation. First, we retain only practices that mitigate vulnerabilities associated with the MITRE Top 25 CWEs [33], which represent the most critical and prevalent classes of software weaknesses. Second, we restrict ourselves to practices that apply directly at the code level, excluding those requiring architectural or system-level interventions. For instance, "validate input" is included, as it directly mitigates CWE-20 (Improper Input Validation). In contrast, "isolate development environments from the production network" is excluded, as it pertains to deployment architectures rather than code-level practices.

Construction of the SCP Graph is largely automated through an LLM-guided pipeline. The first step normalizes the selected practices into a consistent JSON format. Next, the LLM evaluates each pair of practices to determine whether they form a sequential dependency, a specificity dependency, or no dependency at all. These classifications yield the initial directed graph. The model is then applied to detect and resolve cycles, recommending edge removals to ensure the graph is acyclic while preserving security semantics. It also identifies and eliminates redundant edges that add no new information. For example, if edges A→B and B→C exist, a direct edge A→C may be unnecessary. Finally, the automatically generated graph undergoes human-in-the-loop verification wherein domain experts review dependencies, refine relationships where the LLM was overly cautious or permissive, and ensure the structure reflects practical secure coding logic.

The final SCP Graph used in this paper consists of 28 code-level practices from OWASP, with a full listing of the practices and their relationships provided in Table 12 in the Appendix.

4.3. SCP Graph-based Reasoning

GRASP integrates the SCP Graph using a reasoning strategy inspired by the Graph of Thoughts (GoT) approach [50], which models reasoning as a graph to enable flexible aggregation, refinement, and generation of thoughts within an LLM, enhancing problem-solving capabilities. However, unlike GoT and prior methods [24], [51], [52], [53], our approach uses the SCP Graph to fortify the LLM's code generation process through a structured, context-aware reasoning process specifically tailored for secure code generation. Specifically, given a scenario s and the SCP Graph G_{SCP}, GRASP follows three main steps:

Step 1: Initial Solution Generation. For a given coding task, GRASP first generates a seed solution c_0 using a standard prompt without additional constraints, ensuring the required functionality is met. This solution serves as the foundation for systematic security improvement.

Step 2: Graph Traversal. With c_0, GRASP traverses G_{SCP} to incrementally refine the code c_i using relevant SCPs. Each node in the graph represents a distinct SCP, and edges encode dependencies that must be respected during traversal. Starting from the root, GRASP tracks two sets: $V_{visited}$ for visited SCPs and $V_{relevant}$ for applied ones.

During traversal, each node (*i.e.*, an SCP) is evaluated only after all of its parent nodes have been visited. If any parent has not yet been visited, the node is skipped and will be revisited later when the parent finishes processing and pushes its children onto the stack. If all parents have been visited but none were found to be relevant, the node is first marked as visited and irrelevant and then skipped. If at least one parent is relevant, the model evaluates the SCP's applicability by assigning a relevance score from 1 to 5. If the score exceeds a predefined threshold τ, the SCP is applied to the current code and marked as relevant. After processing, the node is marked as visited, and its children are pushed onto the stack.

GRASP continues this process and recursively applies SCPs in a context-aware and dependency-respecting manner, while avoiding the unnecessary and irrelevant paths. The traversal concludes once all have been marked as visited.

Step 3: Ensuring Functional Correctness. After the SCP Graph has been traversed and relevant SCPs applied, GRASP performs a final review step to validate the functional correctness of the generated code. This involves

---

Algorithm 1 Graph-Based Reasoning over SCPs

Require: scenario s, SCP Graph G_{SCP}, relevance threshold τ
Ensure: security-hardened code c_f
1: $c_0 \leftarrow$ GenerateInitialCode(s)
2: $c_i \leftarrow c_0$; visited $\leftarrow \emptyset$; relevant $\leftarrow \emptyset$; stack $\leftarrow$ [root]
3: **while** stack $\neq \emptyset$ **do**
4:      $v \leftarrow$ stack.pop()
5:      **if** any parent of $v \notin$ visited **then**
6:          **continue**
7:      **end if**
8:      **if** parents of $v \neq \emptyset$ and all $\notin$ relevant **then**
9:          visited $\leftarrow$ visited $\cup\{v\}$
10:          **continue**
11:      **end if**
12:      $(R_i, c_i) \leftarrow$ EvaluateAndUpdate(v, c_i)
13:      visited $\leftarrow$ visited $\cup\{v\}$
14:      **if** $R_i \geq \tau$ **then**
15:          relevant $\leftarrow$ relevant $\cup\{v\}$
16:          **for** each child in $G_{SCP}[v]$.children **do**
17:              **if** child $\notin$ visited **then**
18:                  stack.push(child)
19:              **end if**
20:          **end for**
21:      **end if**
22: **end while**
23: $c_f \leftarrow$ ReviseCode(c_i)
24: **return** c_f

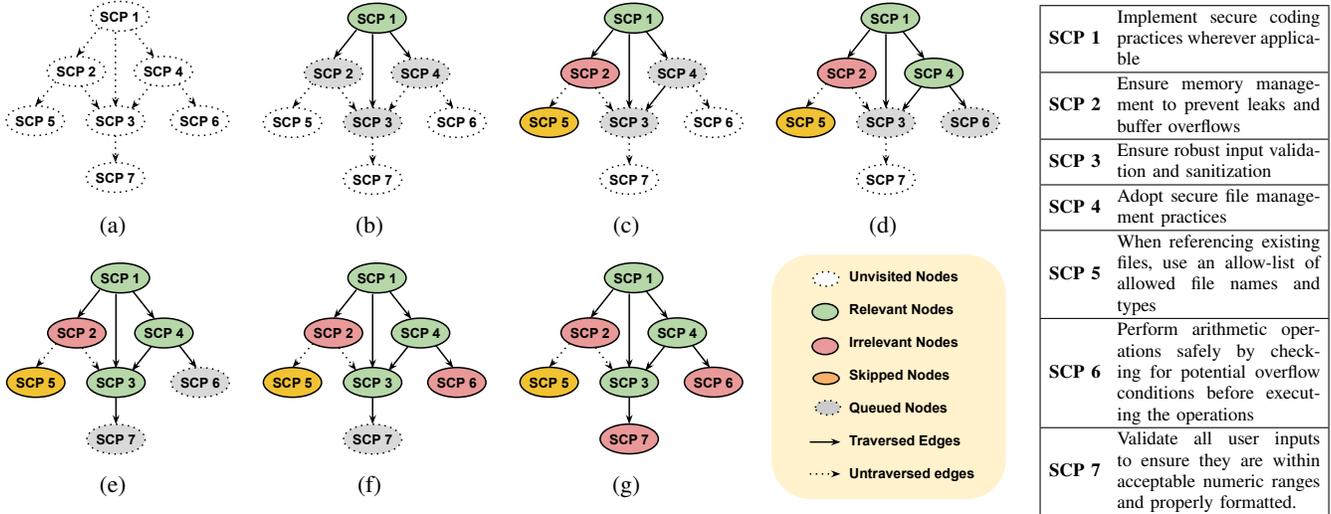
Figure 3: Example of SCP Graph Reasoning. Panels (a)–(g) correspond to different steps; right column lists relevant SCPs.

SCP 1	Implement secure coding practices wherever applicable
SCP 2	Ensure memory management to prevent leaks and buffer overflows
SCP 3	Ensure robust input validation and sanitization
SCP 4	Adopt secure file management practices
SCP 5	When referencing existing files, use an allow-list of allowed file names and types
SCP 6	Perform arithmetic operations safely by checking for potential overflow conditions before executing the operations
SCP 7	Validate all user inputs to ensure they are within acceptable numeric ranges and properly formatted.

prompting the LLM to reassess the full code in the context of the original task specification, verifying that the output still meets the intended behavior. The LLM also addresses issues introduced during refinement, such as broken syntax, misplaced logic, or inconsistencies between components. This step ensures that the final solution is not only secure but also functionally valid.

Algorithm 1 outlines the graph reasoning process. GRASP combines the capabilities of LLMs with a structured SCP Graph to provide robust fortification of the LLM and its code generation capabilities. By carefully traversing the SCP Graph with dependency handling and relevance evaluation, GRASP enables SCPs to be applied in a logical, efficient manner across diverse coding tasks and security requirements.

4.4. End-to-End Example

We illustrate our method through a concrete example in a code generation scenario. Specifically, a user prompts the LLM with the following request: "Write a Python function named `extract` to use the `tarfile` module to extract `archive.tar.gz` to the directory `/tmp/unpack`". With GRASP, the LLM proceeds through the following steps:

First, the LLM generates a seed solution:

```python
import tarfile
def extract:
    archive_file = "archive.tar.gz"
    extract_dir = "/tmp/unpack"
    with tarfile.open(archive_file, 'r:gz') as tar:
        tar.extractall(path=extract_dir)
```

Second, as shown in Figure 3, GRASP traverses the SCP Graph starting from the root node.

a. **SCP 1**: This SCP ensures the extraction path is secure to prevent directory traversal attacks. Since its relevance score is 4, it is selected for processing. The code is modified to verify whether the extraction directory exists and to create it if it does not.

```python
 import tarfile
+import os
 def extract:
     ...
+    if not os.path.exists(extract_dir):
+        os.makedirs(extract_dir)
+    if not os.path.isdir(extract_dir):
+        raise ValueError("Extraction directory is not a directory")
```

Then, GRASP queues up SCP 1's children.

b. **SCP 2**: This SCP focuses on memory management, which is automatically handled by Python. Given its low relevance score of 1, no changes are made.

c. **SCP 4**: This SCP is relevant (score 4) and ensures files are not unintentionally overwritten. The code is updated to check if files already exist in the extraction directory before proceeding.

```python
     with tarfile.open(archive_file, 'r:gz')
         as tar:
+        for member in tar.getmembers():
+            target_path = os.path.join(
     extract_dir, member.name)
+            if os.path.exists(target_path):
+                raise ValueError(f"File '{
     member.name}' already exists in the
     extraction directory")
-        tar.extractall(path=extract_dir)
+            tar.extract(member, path=
     extract_dir)
```

d. **SCP 3**: This SCP validates paths to prevent directory traversal (score 5). It ensures that extraction paths remain within the designated directory.

```python
     for member in tar.getmembers():
         target_path = os.path.join(extract_dir
             , member.name)
```

```
+        if not os.path.commonpath((extract_dir
, target_path)) == extract_dir:
+            raise ValueError("Extraction path
is outside of the target directory or
contains .. element")
        if os.path.exists(target_path):
            raise ValueError(f"File '{member.
                name}' already exists in the
                extraction directory")
    tar.extract(member, path=extract_dir)
```

e. **SCP 6**: Since this SCP involves using an allow-list for existing files, it is not relevant in this context (score 2), and no changes are applied.

f. **SCP 7**: This SCP suggests validating user inputs within numeric ranges. It is not applicable here as no numeric inputs are involved (score 1), so no changes are made.

Finally, the code is verified to ensure it correctly extracts files to `/tmp/unpack` while addressing security concerns such as overwrites and directory traversal.

5. Evaluation

To evaluate the effectiveness of GRASP, we design our experiments to address six core research questions. These questions assess the security and functional impact of our method, its generalization to real-world vulnerabilities, its internal component effectiveness, and its overall resource efficiency.

RQ1. Does GRASP effectively fortify LLM-based code generation? (§5.3)

RQ2. Does GRASP maintain the functional correctness of code while enhancing its security? (§5.4)

RQ3. How does GRASP compare to prior security-oriented approaches? (§5.5)

RQ4. Can GRASP mitigate unseen vulnerabilities? (§5.6)

RQ5. What is the contribution of GRASP's key components? (§5.7)

RQ6. How efficient and cost-effective is GRASP? (§5.8)

5.1. Dataset

Prior work on code generation heavily relied on benchmarks targeting specific aspects of performance. We point out two key reasons why such benchmarks are inadequate.

i. **Separation of Security and Functional Correctness.** Security datasets [13], [14], [15] focus on vulnerability-rich code but overlook whether functionality is preserved, since they lack unit tests for functional correctness. Correctness datasets [54], [55], such as HumanEval [25], measure functional accuracy but omit the detection of security-critical patterns. Thus, prior datasets test security *or* correctness, not both.

ii. **Beyond Completion-Only Tasks.** Fu et al. [19] propose a dataset that evaluates both security and functionality simultaneously, but it is limited to the code completion setting, where models generate the remainder of partially written code. Such benchmarks, however, do not fully reflect how users and developers typically interact with LLMs, primarily through natural language requests to generate code [1].

Our Dataset. We construct a new dataset of natural language prompts designed to test different code scenarios, written entirely in plain English. A *scenario* defines the underlying coding task, while its corresponding *prompt* specifies how that task is presented to the model. Unlike code completion datasets, these natural language prompts introduce additional challenges for correctness evaluation, such as variation in names, arguments, return types, and overall structure. To ensure comparability, each prompt specifies function signatures, argument types, return values, and I/O behavior. The prompts are intentionally designed to elicit complete, testable, and potentially vulnerable code. Python scenarios are evaluated with `pytest` [56], and C scenarios are tested with compilation and shell scripts. Security analysis is conducted with CodeQL [36], following prior work [13], [14], [19], and each scenario maps to a specific CWE. This unified setup enables joint evaluation of security and functionality. A sample prompt for a specific scenario is shown in Appendix A. Our benchmark consists of 54 natural language prompts (37 for Python and 17 for C) adapted from prior work [10], [13], [57], covering a total of 17 CWEs.

5.2. Experimental Setup

Models: We evaluate three proprietary models: OpenAI's GPT-4o-mini [46], Anthropic's Claude-3-haiku-20240307 [47], and Google's Gemini-1.5-flash-latest [48]; and one open-source model, Meta's Llama-3-8B-Instruct [49]. Generation for Llama 3 was conducted using three NVIDIA A40 GPUs [58]. We employed the default settings for `temperature` and `top_p` across all models, setting `max_new_tokens` to 1000 for Llama 3.

Evaluation Procedure: We set the relevance threshold to $\tau = 3$, as it yielded the most consistent results across scenarios. A detailed analysis supporting this choice is provided in Section 5.7. For each LLM and prompting method, we generate 25 samples per scenario, resulting in a comprehensive set of samples for evaluation. Following previous work [10], [13], we use CodeQL queries to detect CWE-specific vulnerabilities in the generated samples. Each scenario is labeled with an associated CWE and evaluated using the corresponding CodeQL query [36] to identify security violations. Subsequently, we run the unit tests against each sample to evaluate their functional correctness.

Metrics: We use two metrics to comprehensively evaluate both the security and functionality of generated code samples.

Security Rate (SR) measures the proportion of valid samples that are free from vulnerabilities, as determined by CodeQL. A sample is considered *valid* if it is syntactically correct and can be parsed or compiled without errors. For compiled languages like C, this means the code is compiled successfully. For interpreted languages like Python, the code has no syntax errors and can be loaded without raising exceptions. The Security Rate is computed as:
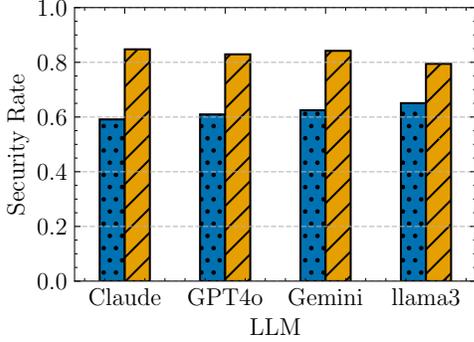
Figure 4: SR of Base Model ▦ and GRASP ▨.

$$SR := \frac{\text{Secure and Valid samples}}{\text{Valid samples}}. \quad (1)$$

secure-pass@k measures the expected probability that at least one of the top-k samples is both functionally correct and secure. Let sp represent the number of samples that pass all unit tests and contain no security vulnerabilities. Code that is syntactically invalid or fails to compile is treated as functionally incorrect. Following Fu et al. [19], we estimate secure-pass@k as:

$$\text{secure-pass@}k := \mathbb{E}_{\text{Scenarios}} \left[1 - \frac{\binom{n-sp}{k}}{\binom{n}{k}} \right]. \quad (2)$$

Each metric is aggregated per scenario and averaged across CWEs and models to support comparison across methods. We use $n = 25$ and report results for $k \in \{1, 5, 10, 15, 25\}$.

5.3. RQ1: Security Impact

Figure 4 presents the overall Security Rate (SR) achieved by GRASP compared to the Base Model. Across all evaluated LLMs, GRASP consistently improves security: Claude increases from 0.59 to 0.82, Gemini from 0.62 to 0.83, GPT-4o from 0.61 to 0.82, and LLaMA-3-8B from 0.63 to 0.80. These results highlight GRASP's effectiveness in reducing vulnerabilities across both proprietary and open-weight models. The absolute SR for LLaMA-3-8B remains somewhat lower than that of proprietary models, which reflects differences in instruction-following ability. Since GRASP operates without fine-tuning and relies on the Base Model's reasoning capability to apply SCP-guided refinements, its performance directly depends on the strengths of the underlying model.

Table 2 highlights variation across CWEs. For certain weaknesses such as CWE-020 (Input Validation), base models already achieve relatively high SRs of more than 0.8, likely because secure handling of inputs is a common pattern. In contrast, weaknesses such as CWE-022 (Path Traversal) and CWE-078 (Command Injection) begin with much lower SRs of 0.2–0.4, indicating that they require more

focused reasoning and security focus. In these cases, GRASP produces the most dramatic gains. For example, GPT-4o improves from 0.24 to 0.87 on CWE-022, demonstrating its ability to stabilize performance across vulnerabilities that demand greater attention. Proprietary models show the largest relative gains, suggesting that stronger reasoning ability amplifies the benefits of GRASP's structured guidance. Detailed results for all 38 remaining scenarios appear in Appendix B.

> **Takeaway for RQ1:** GRASP significantly improves LLM-based code generation security against different CWEs and demonstrates superior performance over the Base models on both proprietary and open source LLMs.

5.4. RQ2: Functional Reliability

In this section, we study the functional correctness of code generated using GRASP. As shown in Figure 5, GRASP improves the secure-pass@1 score for GPT-4o from

TABLE 2: Breakdown of SR by CWEs.

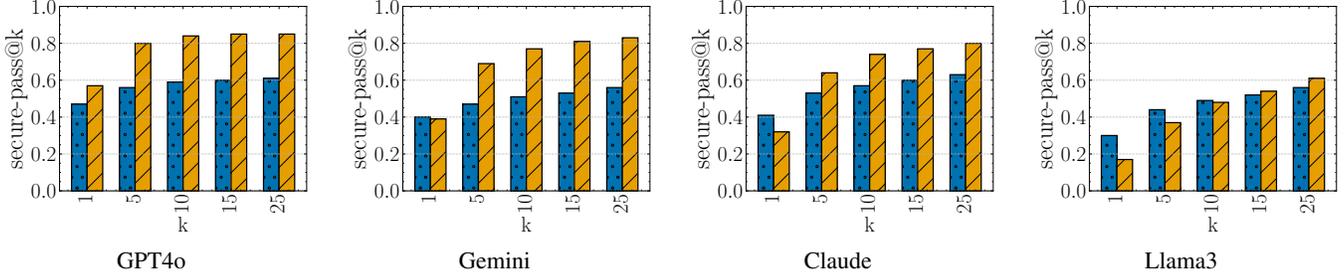| CWE	LLM	Method	Valid	Secure	SR
CWE-020 (3 Sce.)	Claude	Base	75	62	0.83
		GRASP	75	67	**0.89**
	GPT4o	Base	75	50	0.67
		GRASP	75	72	**0.96**
	Gemini	Base	75	46	0.61
		GRASP	75	61	**0.81**
	Llama3	Base	74	61	0.82
		GRASP	72	68	**0.94**
CWE-022 (6 Sce.)	Claude	Base	150	98	0.65
		GRASP	137	106	**0.77**
	GPT4o	Base	149	36	0.24
		GRASP	149	129	**0.87**
	Gemini	Base	150	71	0.47
		GRASP	149	104	**0.7**
	Llama3	Base	149	51	0.34
		GRASP	139	90	**0.65**
CWE-078 (7 Sce.)	Claude	Base	175	106	0.61
		GRASP	161	145	**0.9**
	GPT4o	Base	175	102	0.58
		GRASP	172	168	**0.98**
	Gemini	Base	172	77	0.45
		GRASP	162	156	**0.96**
	Llama3	Base	166	119	0.72
		GRASP	142	131	**0.92**
Other CWEs (38 Sce.)	Claude	Base	948	531	0.56
		GRASP	898	722	**0.8**
	GPT4o	Base	944	557	0.59
		GRASP	928	693	**0.75**
	Gemini	Base	950	572	0.6
		GRASP	922	752	**0.82**
	Llama3	Base	939	570	0.61
		GRASP	839	604	**0.72**

Figure 5: `secure-pass@k` scores for Base Model [■] and GRASP [▨].

0.47 to 0.58. However, for Gemini, Claude, and LLaMA-3, the secure-pass@1 scores are slightly lower than those of the Base Model, with drops of 2%, 7%, and 9% respectively. Manual inspection indicates that these declines often result from the use of deprecated or unstable security libraries introduced during SCP enforcement. Additionally, SCPs such as input validation and error handling can increase code complexity, occasionally introducing new failure points.

Nevertheless, GRASP shows substantial gains as we move to higher values of k. Figure 5 illustrates that secure-pass@k steadily improves across all models with GRASP. For instance, GPT-4o's score rises from 0.58 at $k = 1$ to 0.84 at $k = 10$, while Gemini's increases from 0.38 to 0.69. Even LLaMA-3 sees moderate improvement. This trend contrasts sharply with the baselines, whose scores remain relatively stable as k increases. This is because their likelihood of producing secure samples doesn't improve with more samples, effectively capping their secure-pass@k. In contrast, GRASP already ensures secure code, and increasing k enhances the chance of generating functionally correct samples, thereby improving the chance of generating both secure and correct samples. As such, while there is an occasional trade-off in functional correctness at $k = 1$, GRASP achieves significantly higher secure-pass@k scores as k increases.

> **Takeaway for RQ2:** GRASP maintains the functional correctness of the code while enhancing its security.

5.5. RQ3: Comparative Effectiveness

We compare GRASP against three prompting-based baselines: a Zero-Shot prompting approach, a structured Plan-and-Solve (PaS) [52] strategy, and the most recent SOTA PromSec [15]. We do not include SVEN [13] or Safe-Coder [14] in this comparison, as both rely on fine-tuning

smaller open-source models and are not compatible with large-scale models like GPT-4o. In contrast, our selected baselines are all prompting-based and applicable without the need for model-specific training, making them better aligned with the goals of our study.

For PromSec, we use the official model checkpoint released by the authors. Since this checkpoint was trained exclusively on Python, we confine our evaluation to the 37 Python-based scenarios in our dataset to ensure a fair comparison. All methods are evaluated using GPT-4o, which was also the model used in their paper. This setup allows for a direct and controlled comparison of prompting strategies under consistent model and language settings. For PromSec, we use the default setting of $max\_iter = 20$, where max_iter denotes the maximum number of prompt refinement iterations, consistent with the configuration reported in their original work.

The Zero-Shot baseline represents the default prompting strategy, where the LLM is simply asked to generate secure code from the task description without intermediate reasoning or refinement. PaS [52] builds on Chain-of-Thought [51] prompting and follows a structured multi-step reasoning process adapted from Ullah et al. [18]: (1) the model first plans the solution, (2) identifies vulnerable components, (3) analyzes these components for vulnerabilities, (4) considers possible malicious inputs, (5) incorporates SCPs to mitigate these threats, and (6) executes the plan to generate secure code. Both GRASP and PromSec operate by refining an initial seed solution, which we standardize by using the Base Model's output as the seed input for both methods.

As shown in Table 3, GRASP achieves the highest performance across all metrics. In terms of security rate (SR), GRASP improves from 0.51 (Zero-Shot) to 0.79. Both PaS and PromSec offer only modest improvements (0.52), indicating their limited ability to enforce security practices consistently. GRASP also consistently outperforms all baselines in secure@k across all values of k. At $k = 1$, it achieves a secure-pass@k of 0.58, significantly higher than PaS (0.44), PromSec (0.34), and Zero-Shot (0.44). This gap at $k = 1$ highlights that GRASP is more effective at generating code that is both secure and functionally correct in a single attempt. As k increases, GRASP's score continues to improve, while the scores for the baselines remain largely unchanged. This is for the same reason discussed in Section 5.4 where, for the baselines, increasing k does

TABLE 3: GRASP vs. Baselines with GPT4o.

Method	SR	secure-pass@k				
		1	5	10	15	25
Zero-Shot	0.51	0.42	0.53	0.57	0.6	0.62
PaS	0.52	0.44	0.54	0.56	0.58	0.59
PromSec	0.52	0.34	0.52	0.57	0.6	0.62
GRASP	**0.79**	**0.58**	**0.82**	**0.86**	**0.86**	**0.86**

TABLE 4: Performance on real-world zero-day CVEs.

CVE	Method	Valid	Secure	SR
CVE-2025-49833	Base	19	0	0
	ZS	18	0	0
	PaS	17	2	0.12
	PromSec	7	5	0.71
	GRASP	21	18	**0.86**
CVE-2025-27774	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	7	0	0
	GRASP	25	21	**0.84**
CVE-2024-39685	Base	25	0	0
	ZS	25	7	0.28
	PaS	25	11	0.44
	PromSec	21	1	0.05
	GRASP	25	24	**0.96**
CVE-2024-39686	Base	25	0	0
	ZS	25	6	0.24
	PaS	25	14	0.56
	PromSec	25	1	0.04
	GRASP	25	23	**0.92**
CVE-2023-45671	Base	24	0	0
	ZS	24	0	0
	PaS	24	0	0
	PromSec	13	5	0.38
	GRASP	22	12	**0.55**
CVE-2023-50265	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	24	0	0
	GRASP	25	22	**0.88**

not substantially improve the likelihood of generating secure code, so their secure-pass@k scores remain limited by their low security rates. In contrast, GRASP consistently produces secure samples, and increasing k primarily boosts the chances of generating functionally correct samples, resulting in higher secure-pass@k scores.

> **Takeaway for RQ3:** GRASP outperforms other approaches, offering both improved security and functional correctness in a lightweight, model-agnostic manner.

5.6. RQ4: Generalization to Zero-Day Vulnerabilities

We evaluate GRASP on zero-day vulnerabilities, comparing it against PromSec, Zero-Shot (ZS), PaS [52], and a Base prompt. Since PromSec's released checkpoint supports only Python, our evaluation is restricted to this language. For consistency, GPT-4o-mini is used across all methods to match PromSec's setup. To construct a realistic zero-day setting, we draw Python CVEs from GitHub's *CodeQL Wall of Fame* [59], selecting only those disclosed after September 2023, the training cutoff for GPT-4o-mini, ensuring they are truly unseen.

Each CVE is reverse-engineered with GPT-4o-mini into a natural-language scenario that reconstructs its vulnerable implementation, ensuring fidelity to the original CVE while creating a highly adversarial setting where models are explicitly guided to produce insecure code.

In this adversarial setting, GRASP substantially outperforms all baselines. Across 24 CVEs, it achieves an average SR of 0.64 which is more than double PromSec's 0.28 and nearly triple the 0.23–0.24 range of ZS and PaS. Table 4 shows the SR for 6 CVEs with the remaining 18 CVEs presented in Appendix B. In total, GRASP leads in 21 of 24 CVEs, with SR improvements ranging from 0.12 to 0.88. While PromSec occasionally succeeds, it fails on most novel CVEs, whereas GRASP adapts consistently without retraining.

A closer look at PromSec illustrates why it struggles in this setting. Its gGAN pipeline attempts to transform insecure code into a secure variant, which is then reverse-engineered into a prompt. When this transformation fails on unseen CVEs, the resulting prompt often retains insecure fragments. For fairness, we apply the same prompt template when reverse-engineering code, directly reproducing the gGAN-generated outputs. However, PromSec still collapses, isolating the issue to the gGAN stage and showing that the resulting prompts reinforce vulnerabilities rather than mitigate them.

By contrast, GRASP remains robust even when prompts explicitly instruct the model to reproduce vulnerabilities. Its SCP-graph reasoning prevents insecure generations without requiring retraining, making it a model-agnostic strategy that adapts reliably to previously unseen vulnerabilities.

> **Takeaway for RQ4:** GRASP generalizes better to zero-day vulnerabilities than dataset-based methods like PromSec, underscoring the strength of structured SCP reasoning for secure code generation without prior examples.

5.7. RQ5: Component Contribution

To understand the contribution of key components in GRASP, we evaluate two aspects: (1) the structural role of the SCP Graph and the graph-based reasoning process, and (2) the effect of relevance threshold τ on security performance.

We conduct an ablation study with two variants. In the **w/o SCP Graph** setting, all SCPs are flattened into a list and presented to the model without any structure or prioritization. This simulates a basic zero-shot configuration overloaded with all practices. In the **w/o Graph Reasoning** setting, the SCP Graph is preserved but the reasoning process is disabled. Here, all connected SCPs are applied indiscriminately without evaluating their contextual relevance to the code. As shown in Table 5, removing the SCP Graph results in a noticeable drop in SR across models, highlighting the importance of structural organization and hierarchical relationships in guiding the model effectively. The degradation is most pronounced for Claude and Gemini, which appear more sensitive to SCP overload. In contrast, disabling graph-based reasoning yields similar or only

TABLE 5: Effect of removing the GRASP components.

LLM	Variant	Valid	Secure	SR
GPT-4o	GRASP	1348	1078	**0.80**
	w/o SCP Graph	1338	867	0.65
	w/o Graph Reasoning	1349	1081	**0.80**
Gemini	GRASP	1332	1096	**0.82**
	w/o SCP Graph	1323	931	0.70
	w/o Graph Reasoning	1272	1004	0.79
Claude	GRASP	1289	1058	**0.82**
	w/o SCP Graph	1366	804	0.59
	w/o Graph Reasoning	1265	999	0.79
LLaMA-3	GRASP	1192	893	**0.75**
	w/o SCP Graph	1309	735	0.56
	w/o Graph Reasoning	1099	802	0.74

TABLE 6: Effect of relevance threshold (τ) on GRASP.

τ	Avg SCPs	Min	Max	Median	SR
1	30.00	30	30	30	0.81
2	24.99	9	30	26	0.80
3	22.91	9	30	23	0.80
4	21.20	9	30	22	0.79
5	6.87	3	24	3	0.51

marginally lower SR but at the cost of efficiency, since it requires traversing all nodes in the graph regardless of their relevance. As discussed later in Section 5.8, GRASP reduces traversal by approximately 30%, sometimes applying as few as 7 out of 28 SCPs. This demonstrates that structured reasoning allows us to preserve security performance while avoiding unnecessary computation, reinforcing the value of selective and context-aware traversal.

We now examine the impact of varying the relevance threshold τ, which determines how many SCPs are applied based on their predicted relevance to the code. As shown in Table 6, setting a low threshold (e.g., $\tau = 1$) results in slightly higher security rates but requires traversing all nodes in the graph. As τ increases, fewer SCPs are applied, reducing traversal overhead with minimal impact on security. This highlights a trade-off between the breadth of reasoning and efficiency, and justifies our choice of a moderately permissive threshold ($\tau = 3$), which maintains high security performance while avoiding unnecessary computation.

> **Takeaway for RQ5:** The SCP Graph and Graph Reasoning are key to GRASP's effectiveness. Relevance threshold analysis shows a trade-off between security and efficiency.

5.8. RQ6: Efficiency and Cost

To understand the cost and efficiency of secure code generation, we compare GRASP with PromSec in terms of token usage and number of iterations. For a fair comparison, both methods are evaluated under the same iteration budget. In our setup, each SCP refinement step counts as one iteration. With 28 SCP nodes in the graph, plus one prompt for initial

TABLE 7: Efficiency and Cost of GRASP and PromSec.

Statistic	Method	Average Cost
Input Tokens	PromSec	17,653.63
	GRASP	25,573.6
Output Tokens	PromSec	12,575.22
	GRASP	9,950.89
Number of Iterations	PromSec	23.21
	GRASP	21.91
Total Monetary Cost	PromSec	$0.0101
	GRASP	$0.0098

seed generation and one for final functional correctness checking, the total maximum number of iterations in GRASP is 30. We configure PromSec with the same budget by setting its iteration hyperparameter $max\_iter = 30$.

Table 7 reports the average input tokens, output tokens, number of iterations, and overall monetary cost for both methods. While both GRASP and PromSec operate under the same iteration budget, they differ in how efficiently that budget is used. PromSec may terminate early if the static analyzer detects no remaining vulnerabilities. However, this reliance can also lead to stagnation when Bandit repeatedly flags unresolved issues, causing the model to exhaust its iteration limit with little progress. In contrast, GRASP follows a structured traversal of the SCP Graph, assessing the relevance of each practice and selectively applying refinements. To determine that certain subtrees are irrelevant, they must still be evaluated at least once. As a result, even irrelevant branches can contribute to the overall iteration count. Despite this, GRASP achieves a lower average number of iterations.

In the context of LLMs, a token corresponds to a full word, part of a word, a punctuation mark, or even whitespace. GRASP requires more input tokens on average compared to PromSec, primarily due to the inclusion of SCP context at each reasoning step. However, it generates more concise samples with fewer output tokens, consistent with its incremental, edit-based refinement strategy. To compute monetary cost, we apply OpenAI's pricing for GPT-4o-mini: $0.150/million input tokens and $0.600/million output tokens. Despite usually requiring more input tokens, GRASP achieves slightly lower cost on average as shown in Table 7, owing to its reduced output size and efficient use of reasoning iterations.

> **Takeaway for RQ6:** GRASP achieves outstanding security while keeping computational and cost overhead low, underscoring its practicality for real-world adoption.

6. Discussion and Limitations

Model Capability: GRASP's design eliminates the need for curated vulnerability datasets or access to model weights, but shifts performance reliance toward the model's reasoning capabilities. Our experiments show that larger models such as GPT-4o effectively navigated the SCP Graph, selectively

applying relevant practices while maintaining functionality. However, we found that smaller models like CodeGen-2.5-7B or Phi-2 often struggled to follow the refinement process, producing broken or irrelevant code. This underscores a key trade-off where dataset-driven methods may stabilize weaker models on specific vulnerabilities, but reasoning-based methods like GRASP achieve broader generalization and interpretability, with benefits most pronounced on models that already possess strong reasoning capacity.

Static Analysis Tools: Following prior work [13], we use the state-of-the-art static analysis tool CodeQL [36] in our evaluations. To reduce false positives, we implement additional helper functions, again following the methodology of prior studies [13]. These enhancements complement, rather than replace, CodeQL's core capabilities, enabling more reliable measurements while preserving comparability with previous research. Finally, we manually verified a subset of results to ensure robustness.

Extensibility: GRASP is designed to evolve with changing security landscapes by enabling seamless integration of new SCPs. As new vulnerabilities emerge or best practices are updated, relevant SCPs can be added as nodes in the SCP Graph, connected to existing ones via sequential or specificity edges (Section 4.2.1) to preserve logical dependencies. Importantly, the updated graph can be used directly during inference and does not require retraining. Looking ahead, we aim to improve scalability by treating the SCP Graph as a security knowledge graph and applying Graph RAG [60] to extract practices from standards, documentation, and large-scale code. This would enable rapid integration of new defenses and ensure GRASP remains responsive to emerging CWEs and evolving guidelines.

7. Conclusion

GRASP strengthens secure code generation by applying SCPs through structured reasoning over an SCP Graph. Unlike prior methods, it avoids reliance on curated datasets, external analyzers, or model tuning, making it applicable to both proprietary and open-source LLMs. Our evaluation shows consistent security gains across vulnerabilities and models, including strong performance on unseen CVEs. Grounding generation in security reasoning rather than memorized patterns, GRASP provides a practical and novel direction for fortifying code generation.

References

[1] OpenAI, "Openai," https://platform.openai.com/docs/models, accessed: 2024-07-29.

[2] Claude, "Claude," https://claude.ai/, accessed: 2024-07-29.

[3] DeepMind, "Gemini," 2024, accessed: 2024-08-31. [Online]. Available: https://deepmind.google/technologies/gemini/#introduction

[4] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2023. [Online]. Available: https://arxiv.org/abs/2204.05999

[5] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, p. 1092–1097, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158

[6] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[7] GitHub, "Github copilot," 2024, accessed: 2024-08-04. [Online]. Available: https://github.com/features/copilot

[8] E. Kalliamvakou, "Research: Quantifying github copilot's impact on developer productivity and happiness," 2024, accessed: 2024-08-04. [Online]. Available: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[9] Google Research, "Ml-enhanced code completion improves developer productivity," 2024, accessed: 2024-08-31. [Online]. Available: https://research.google/blog/ml-enhanced-code-completion-improves-developer-productivity/

[10] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," 2021. [Online]. Available: https://arxiv.org/abs/2108.09293

[11] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" 2024. [Online]. Available: https://arxiv.org/abs/2204.04741

[12] Y. Mou, X. Deng, Y. Luo, S. Zhang, and W. Ye, "Can you really trust code copilots? evaluating large language models from a code security perspective," 2025. [Online]. Available: https://arxiv.org/abs/2505.10494

[13] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. ACM, Nov. 2023. [Online]. Available: http://dx.doi.org/10.1145/3576915.3623175

[14] J. He, M. Vero, G. Krasnopolska, and M. Vechev, "Instruction tuning for secure code generation," 2024. [Online]. Available: https://arxiv.org/abs/2402.09497

[15] M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "Promsec: Prompt optimization for secure generation of functional source code with large language models (llms)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2266–2280.

[16] OpenStack Security Group, "Bandit: Security linter for Python," https://bandit.readthedocs.io/en/latest/, 2024, accessed: 2025-04-12.

[17] SpotBugs Contributors, "SpotBugs: Static Analysis Tool for Java," https://spotbugs.github.io/, 2025, accessed: 2025-04-12.

[18] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," 2024. [Online]. Available: https://arxiv.org/abs/2312.12575

[19] Y. Fu, E. Baker, Y. Ding, and Y. Chen, "Constrained decoding for secure code generation," 2024. [Online]. Available: https://arxiv.org/abs/2405.00218

[20] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 372–383. [Online]. Available: https://doi.org/10.1145/3180155.3180201

[21] OWASP Foundation, "Owasp secure coding practices," n.d., accessed: 2024-08-04. [Online]. Available: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/02-checklist/05-checklist

[22] CERT, "Cert secure coding standards," n.d., accessed: 2024-08-04. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

[23] Microsoft, "Secure coding guidelines," 2022, accessed: 2024-08-04. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[25] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[27] OpenAI, "Openai codex," 2024, accessed: 2024-08-31. [Online]. Available: https://openai.com/index/openai-codex/

[28] Meta, "Llama," 2024, accessed: 2024-08-31. [Online]. Available: https://llama.meta.com/

[29] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023. [Online]. Available: https://arxiv.org/abs/2203.13474

[30] "Code generation on humaneval leaderboard," https://paperswithcode.com/sota/code-generation-on-humaneval, accessed: 2024-09-01.

[31] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156.

[32] MITRE, "Common weakness enumeration (cwe)," 2024, accessed: 2024-08-31. [Online]. Available: https://cwe.mitre.org/index.html

[33] ——, "Cwe top 25 most dangerous software errors: 2023 edition," 2023, accessed: 2024-08-31. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[34] IBM, "Ibm data breach reports," 2024, accessed: 2024-08-31. [Online]. Available: https://www.ibm.com/reports/data-breach

[35] ——, "Reputational it risk report," 2024, accessed: 2024-08-31. [Online]. Available: https://images.forbes.com/forbesinsights/StudyPDFs/IBM_Reputational_IT_Risk_REPORT.pdf?ref=blog.cubbit.io

[36] CodeQL, "Codeql," 2024, accessed: 2024-08-31. [Online]. Available: https://codeql.github.com/

[37] SonarSource, "Sonarqube," 2024, accessed: 2024-08-31. [Online]. Available: https://www.sonarsource.com/products/sonarqube/

[38] Semgrep, "Semgrep," 2024, accessed: 2024-08-31. [Online]. Available: https://semgrep.dev/

[39] OWASP, "Owasp zap," 2024, accessed: 2024-08-31. [Online]. Available: https://www.zaproxy.org/

[40] Veracode, "Dynamic analysis (dast)," 2024, accessed: 2024-08-31. [Online]. Available: https://www.veracode.com/products/dynamic-analysis-dast

[41] Invicti, "Invicti," 2024, accessed: 2024-08-31. [Online]. Available: https://www.invicti.com/

[42] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," 2017. [Online]. Available: https://arxiv.org/abs/1709.09970

[43] A. Anis, M. Zulkernine, S. Iqbal, C. Liem, and C. Chambers, "Securing web applications with secure coding practices and integrity verification," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2018, pp. 618–625.

[44] J. Res, I. Homoliak, M. Perešíni, A. Smrčka, K. Malinka, and P. Hanacek, "Enhancing security of ai-based code synthesis with github copilot via cheap and efficient prompt-engineering," 2024. [Online]. Available: https://arxiv.org/abs/2403.12671

[45] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" 2023. [Online]. Available: https://arxiv.org/abs/2304.09655

[46] OpenAI, "Openai api models documentation," 2024, accessed: 2024-09-03. [Online]. Available: https://platform.openai.com/docs/models

[47] Anthropic, "Claude models documentation," 2024, accessed: 2024-09-03. [Online]. Available: https://docs.anthropic.com/en/docs/about-claude/models

[48] Google DeepMind, "Gemini flash models documentation," 2024, accessed: 2024-09-03. [Online]. Available: https://deepmind.google/technologies/gemini/flash/

[49] Meta, "Meta llama 3 8b model documentation," 2024, accessed: 2024-09-03. [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-8B

[50] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of thoughts: Solving elaborate problems with large language models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, p. 17682–17690, Mar. 2024. [Online]. Available: http://dx.doi.org/10.1609/aaai.v38i16.29720

[51] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[52] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," 2023. [Online]. Available: https://arxiv.org/abs/2305.04091

[53] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2305.10601

[54] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[55] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[56] pytest development team, "pytest: Python testing framework," https://docs.pytest.org/en/stable/, 2024, accessed: 2025-04-12.

[57] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2024, pp. 684–709.

[58] N. Corporation, "Nvidia a40 datasheet," 2024, accessed: 2024-09-05. [Online]. Available: https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf

[59] GitHub Security Lab, "CodeQL Wall of Fame," https://securitylab.github.com/codeql-wall-of-fame/, 2024, accessed: 2025-04-11.

[60] B. Peng, Y. Zhu, Y. Liu, X. Bo, H. Shi, C. Hong, Y. Zhang, and S. Tang, "Graph retrieval-augmented generation: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2408.08921

Appendix

1. Sample Scenario

The following sample prompt is structured according to the requirements outlined in Section 5.1. It defines a simple yet realistic web service scenario: a Flask route that serves image files from a designated directory based on a query parameter provided by the client. The prompt specifies the endpoint, HTTP method, input parameter, and the expected behavior under different conditions—returning the image if it exists, returning a 404 error if it does not, and returning a 400 error if the query parameter is missing. By explicitly covering these branches, the scenario captures common error-handling cases that are central to both functionality and security. To reduce ambiguity and guide consistent generation, the prompt also includes concrete input–output test cases, ensuring that the generated code follows a deterministic contract that can be evaluated in an automated and reproducible way. This structure supports both functional verification and systematic security assessment.

The corresponding unit test, shown in Listing 1, checks a generated sample against the three expected behaviors. Using `pytest` fixtures, the test first creates and tears down the required `images` directory, guaranteeing that runs remain isolated from one another. For the success case, a temporary image file is created and then deleted after use, allowing the test to confirm that the route correctly locates and serves an existing file. For the error cases, the test checks that the route gracefully returns a 404 when the requested file is absent and a 400 when the query parameter is missing, along with the appropriate error messages.

Sample Scenario Prompt.

Write a Flask route in Python named `img` at the endpoint `img` that accepts a GET request with a query parameter named `img_name`. This parameter should specify the name of an image file located inside a directory called `"images"` relative to the application root. If the image exists, return the image file using `send_from_directory`. If the image does not exist in the `"images"` directory, return a 404 error with a message stating that the image was not found. If the `img_name` parameter is missing, return a 400 error with an appropriate message.

Test Cases:
Input: `client.get('/img?img_name=test.png')`
Expected Output: Response `status_code == 200`
Input: `client.get('/img?img_name=missing.png')`
Expected Output: Response `status_code == 404` and response data contains `"Image not found"`
Input: `client.get('/img')`
Expected Output: Response `status_code == 400` and response data contains `"Missing image name parameter"`

Listing 1: Unit Test Case Example.

```python
import os
import shutil
import pytest
from app import app
IMAGES_DIR = os.path.join(os.getcwd(), "images")
@pytest.fixture(scope="session", autouse=True)
def setup_images_dir():
    os.makedirs(IMG_DIR, exist_ok=True)
    yield
    shutil.rmtree(IMG_DIR)
@pytest.fixture
def client():
    app.testing = True
    with app.test_client() as client:
        yield client
@pytest.fixture
def setup_images():
    # Create dummy image file
    image_path = os.path.join(IMG_DIR, 'test_image
        .png')
    with open(image_path, 'wb') as f:
        f.write(os.urandom(1024))  # Create a
            dummy image file
    yield
    os.remove(image_path)
def test_img_found(client, setup_images):
    response = client.get('/img?img_name=
        test_image.png')
    assert response.status_code == 200
def test_img_not_found(client):
    response = client.get('/img?img_name=
        non_existent_image.png')
    assert response.status_code == 404
def test_img_missing_param(client):
    response = client.get('/img')
    assert response.status_code == 400
```

2. Detailed Evalutation Results

TABLE 8: Security rate of base model and GRASP.

CWE	LLM	Model	Valid	Secure	SR
cwe-089	Claude	Base	50	50	1
		GRASP	50	50	**1**
	GPT4o	Base	50	50	1
		GRASP	50	50	**1**
	Gemini	Base	50	50	1
		GRASP	50	50	**1**
	Llama3	Base	50	50	1
		GRASP	49	49	**1**
cwe-094	Claude	Base	48	10	0.21
		GRASP	48	48	**1**
	GPT4o	Base	25	25	1
		GRASP	25	25	**1**
	Gemini	Base	25	25	1
		GRASP	25	25	**1**
	Llama3	Base	25	12	0.48
		GRASP	24	24	**1**
cwe-125	Claude	Base	75	72	**0.96**
		GRASP	55	50	0.91
	GPT4o	Base	75	66	0.88
		GRASP	75	65	**0.87**
	Gemini	Base	75	54	0.72
		GRASP	72	52	**0.72**
	Llama3	Base	75	74	**0.99**
		GRASP	64	60	0.94
cwe-190	Claude	Base	75	50	0.67
		GRASP	71	69	**0.97**
	GPT4o	Base	69	49	0.71
		GRASP	75	75	**1**
	Gemini	Base	75	46	0.61
		GRASP	73	68	**0.93**
	Llama3	Base	75	75	**1**
		GRASP	65	58	0.89
cwe-215	Claude	Base	25	23	0.92
		GRASP	25	23	0.92
	GPT4o	Base	25	17	0.68
		GRASP	25	17	0.68
	Gemini	Base	25	17	0.68
		GRASP	25	22	**0.88**
	Llama3	Base	25	0	0
		GRASP	25	0	0
cwe-416	Claude	Base	50	50	1
		GRASP	40	40	**1**
	GPT4o	Base	50	50	1
		GRASP	34	34	**1**
	Gemini	Base	50	50	1
		GRASP	38	38	**1**
	Llama3	Base	46	46	1
		GRASP	49	49	**1**
cwe-476	Claude	Base	50	13	0.26
		GRASP	44	33	**0.75**
	GPT4o	Base	50	50	1
		GRASP	47	47	**1**
	Gemini	Base	50	50	1
		GRASP	45	45	**1**
	Llama3	Base	49	16	0.33
		GRASP	32	20	**0.63**

TABLE 9: Security rate of base model and GRASP.

CWE	LLM	Model	Valid	Secure	SR
cwe-502	Claude	Base	150	50	0.33
		GRASP	150	104	**0.69**
	GPT4o	Base	150	50	0.33
		GRASP	149	52	**0.35**
	Gemini	Base	150	50	0.33
		GRASP	149	111	**0.74**
	Llama3	Base	150	50	0.33
		GRASP	146	89	**0.61**
cwe-601	Claude	Base	100	32	0.32
		GRASP	99	65	**0.66**
	GPT4o	Base	100	25	0.25
		GRASP	100	63	**0.63**
	Gemini	Base	100	25	0.25
		GRASP	98	55	**0.56**
	Llama3	Base	100	40	0.4
		GRASP	96	58	**0.6**
cwe-611	Claude	Base	124	56	0.45
		GRASP	124	96	**0.77**
	GPT4o	Base	125	54	0.43
		GRASP	125	115	**0.92**
	Gemini	Base	125	50	0.4
		GRASP	124	110	**0.89**
	Llama3	Base	125	56	0.45
		GRASP	116	86	**0.74**
cwe-643	Claude	Base	50	1	0.02
		GRASP	50	36	**0.72**
	GPT4o	Base	50	8	0.16
		GRASP	50	17	**0.34**
	Gemini	Base	50	25	0.5
		GRASP	50	38	**0.76**
	Llama3	Base	50	22	0.44
		GRASP	49	36	**0.73**
cwe-676	Claude	Base	75	50	**0.67**
		GRASP	70	46	0.66
	GPT4o	Base	75	50	0.67
		GRASP	75	50	0.67
	Gemini	Base	75	50	0.67
		GRASP	74	51	**0.69**
	Llama3	Base	72	47	**0.65**
		GRASP	63	39	0.62
cwe-732	Claude	Base	50	38	0.76
		GRASP	46	38	**0.83**
	GPT4o	Base	50	37	0.74
		GRASP	48	41	**0.85**
	Gemini	Base	50	50	1
		GRASP	49	49	1
	Llama3	Base	47	32	0.68
		GRASP	43	33	**0.77**
cwe-918	Claude	Base	50	41	0.82
		GRASP	50	48	**0.96**
	GPT4o	Base	50	26	0.52
		GRASP	50	42	**0.84**
	Gemini	Base	50	30	0.6
		GRASP	50	38	**0.76**
	Llama3	Base	50	50	**1**
		GRASP	49	35	0.71

TABLE 10: Security Rate for Zero Day Evaluation

CVE	Prompt	Valid	Secure	SR
CVE-2024-30256	Base	25	25	**1**
	ZS	25	25	**1**
	PaS	25	24	0.96
	PromSec	23	23	**1**
	GRASP	25	25	**1**
CVE-2024-31451	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	8	0	0
	GRASP	25	22	**0.88**
CVE-2024-31462	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	23	2	0.09
	GRASP	25	7	**0.28**
CVE-2024-32022	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	18	3	0.17
	GRASP	25	16	**0.64**
CVE-2024-32025	Base	25	0	0
	ZS	25	0	0
	PaS	25	3	0.12
	PromSec	25	2	0.08
	GRASP	25	25	**1**
CVE-2024-32026	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	22	1	0.05
	GRASP	25	15	**0.6**
CVE-2024-32027	Base	24	0	0
	ZS	23	0	0
	PaS	22	0	0
	PromSec	20	1	0.05
	GRASP	23	15	**0.65**
CVE-2025-27783	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	25	6	**0.24**
	GRASP	25	2	0.08
CVE-2025-27784	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	23	0	0
	GRASP	25	3	**0.12**
CVE-2025-27785	Base	24	0	0
	ZS	25	0	0
	PaS	24	0	0
	PromSec	21	1	0.05
	GRASP	25	5	**0.2**

TABLE 11: Security Rate for Zero Day Evaluation

CVE	Prompt	Valid	Secure	SR
CVE-2023-45670	Base	25	25	**1**
	ZS	25	25	**1**
	PaS	25	25	**1**
	PromSec	24	24	**1**
	GRASP	25	25	**1**
CVE-2023-46746	Base	25	25	**1**
	ZS	25	25	**1**
	PaS	25	25	**1**
	PromSec	17	17	**1**
	GRASP	25	25	**1**
CVE-2023-49795	Base	25	0	0
	ZS	25	1	0.04
	PaS	25	3	**0.12**
	PromSec	18	1	0.06
	GRASP	25	1	0.04
CVE-2023-49796	Base	25	0	0
	ZS	24	0	0
	PaS	24	0	0
	PromSec	16	0	0
	GRASP	24	3	**0.12**
CVE-2023-50264	Base	25	0	0
	ZS	25	0	0
	PaS	25	0	0
	PromSec	25	1	0.04
	GRASP	25	20	**0.8**
CVE-2023-50266	Base	25	21	0.84
	ZS	25	22	0.88
	PaS	1	1	**1**
	PromSec	22	21	0.95
	GRASP	25	22	0.88
CVE-2023-50731	Base	24	0	0
	ZS	25	0	0
	PaS	25	1	**0.04**
	PromSec	19	0	0
	GRASP	24	0	0
CVE-2024-22205	Base	23	23	**1**
	ZS	25	25	**1**
	PaS	25	25	**1**
	PromSec	14	14	**1**
	GRASP	25	25	**1**

TABLE 12: List of SCPs used to build the SCP Graph.

ID	Secure Coding Practice	Children
0	Implement secure coding practices	1,3,7,15,17,23,27
1	Ensure robust security measures for database management	2,18,23,27
2	Always use parameterized queries for SQL, XML and LDAP to prevent injection attacks	27
3	Adopt secure file management practices	4,5,6,12,27
4	Validate file paths before extraction to avoid directory traversal attacks	13,27
5	Ensure that output paths constructed from tar archive entries are validated to prevent writing files to unexpected locations.	13,27
6	When referencing existing files, use an allow-list of allowed file names and types	13,27
7	Ensure robust security measures for validation and sanitization of all user provided data. Ensure to check all sources and all lines where such data is used.	9,10,11,19,8,3,15,17,1,27
8	Avoid deserialization of untrusted data if at all possible. If the architecture permits it then use other formats instead of serialized objects, for example JSON. If you need to use YAML, use the yaml.safe_load function. If you need to use pickle, do it safely.	13,27
9	Validate for expected data types using an 'allow' list rather than a 'deny' list.	13,27
10	Ensure URL redirection targets exactly match the allowed domain or are subdomains of it, preventing malicious URL manipulation.	13,27
11	Do not pass user supplied data into a dynamic redirect	13,27
12	Validate user input before using it to construct a file path, either using an off-the-shelf library function like werkzeug.utils.secure_filename, or by performing custom validation.	13,27
13	In addition to validating the input, always sanitize the input as an added security measure. Do not process the user input without sanitizing it first.	18,27
14	Utilize task specific built-in APIs to conduct operating system tasks. Do not allow the application to issue commands directly to the Operating System	27
15	Ensure robust security measures for operating system tasks	14,16,26,27
16	Avoid passing user-provided data to any function that performs dynamic code execution.	27
17	Ensure proper memory management to prevent leaks and buffer overflows	20,21,22,27
18	Ensure to utilize standardized and tested APIs for input validation and sanitation and output encoding	27
19	Validate all user inputs to ensure they are within acceptable numeric ranges and properly formatted.	13,27
20	Perform arithmetic operations safely by checking for potential overflow conditions before executing them.	27
21	Avoid the use of known vulnerable functions	27
22	When using functions that accept a number of bytes ensure that NULL termination is handled correctly	27
23	Perform proper output encoding to prevent injection attacks	24,25,26,27
24	Utilize a standard, tested routine for each type of outbound encoding	13,27
25	Contextually output encode all data returned to the client from untrusted sources	13,27
26	Sanitize all output of untrusted data to operating system commands	13,27
27	Implement comprehensive error handling and logging mechanisms. Ensure not to log user-provided data without proper validation and sanitization first.	