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

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 [10, 13, 15, 31, 32, 44]. 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 [17, 18]. 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 [21].

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 [4, 50]. More recently, Mou et al. [41] 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 [23] applied prefix tuning to improve generation security. SafeCoder [24] extended this approach using instruction tuning to provide more flexible control over secure code generation. PromSec [42] introduced prompt-level optimization through graph-based adversarial training, leveraging curated vulnerability datasets and external static analyzers such as Bandit [47] and SpotBugs [56] 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 yet complementary 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 [16, 57]. 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* [34]. 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) [8, 38, 49], which provide the structured discipline needed to translate abstract knowledge into consistent implementation. Empirical studies [34] 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, 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. Brown et al. [7] demonstrated the potential of large-scale language models to perform various tasks, including code generation, with minimal task-specific fine-tuning. Chen et al. [9] further explored the capabilities of LLMs trained specifically in code, highlighting their potential and limitations. The development of CodeBERT by Feng et al. [14] 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 [44], Anthropic's Claude [10], and Google's Gemini [13] offer advanced code generation features with strong instruction-following abilities. They are designed to understand and generate code across various programming languages. GitHub Copilot [17], built on OpenAI's Codex [46], is specifically tailored for code completion and generation tasks within development environments. In contrast, open-source models like Meta's LLama3 [36] and Salesforce's CodeGen [43] 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 [1].

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 [33].

The Common Weakness Enumeration (CWE) framework [40], 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 [39]. 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 [25, 26]. Effective detection and mitigation of these vulnerabilities involve techniques such as static code analysis [11, 54, 55] and dynamic code analysis [27, 48, 58], 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 [2, 35]. 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 [49], CERT Secure Coding Standards [8], and Microsoft Secure Coding Guidelines [38] 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. [23] 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. [24] 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 [53].

Recent work by PromSec [42] 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 [18, 21]. Although LLMs have made significant strides in code generation, they still present substantial security risks [30, 53]. Studies have shown that AI tools for code generation like GitHub Copilot generate vulnerable code about 40% of the time [4, 50]. Khoury et al. found that, out of 21 code generation tasks, ChatGPT [44] produced secure code for only 5 tasks that met security standards [30].

In our study, we assessed the tendency of LLMs to generate vulnerable code. To do this, we evaluated OpenAI's GPT-4o-mini [45], Anthropic's Claude-3-haiku-20240307 [3], Google's Gemini-1.5-flash-latest [20] and Meta's Llama-3-8B-Instruct [37]. 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 [11], 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 Gem-

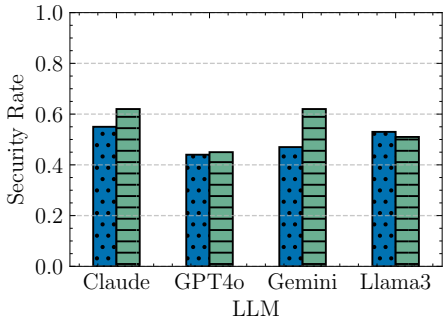


Figure 1: SR of Base Model and Zero-Shot Method.

ini'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 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 [23] and SafeCoder [24] harden LLMs through prefix-tuning and instruction tuning, respectively, with both methods requiring large, curated datasets of vulnerability-fix pairs. PromSec [42], 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.

Table 1: Comparison of Secure Code Generation Approaches.

Property	SVEN	SafeCoder	PromSec	GRASP
Closed-source LLM compatible	○	○	●	●
No training needed	○	○	◐	●
No vuln 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.

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 [23] and SafeCoder [24], 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 [42] requires iterative prompt refinement using external static analyzers such as Bandit [47] or SpotBugs [56]. 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 in-

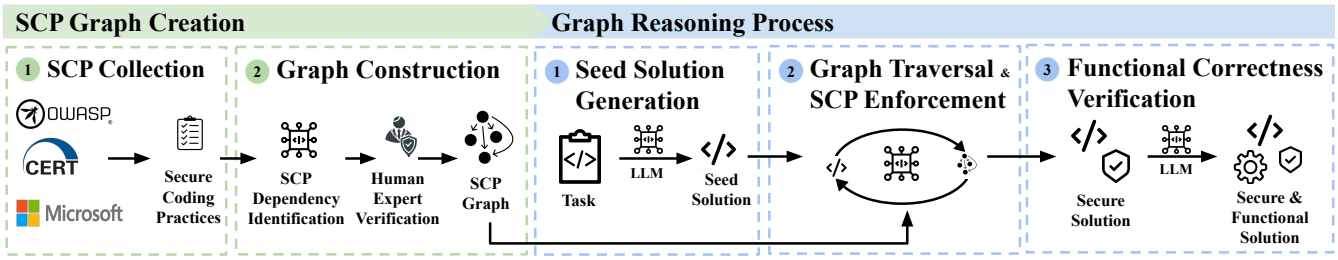


Figure 2: Overview of GRASP.

stances, 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 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 [49], CERT [8], and Microsoft's secure coding standards [38]. 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 [39], 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 complete process is detailed in Appendix D.

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 15 in Appendix D.

4.3 SCP Graph-based Reasoning

GRASP integrates the SCP Graph using a reasoning strategy inspired by the Graph of Thoughts (GoT) approach [6], 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 [7, 59–61], 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.

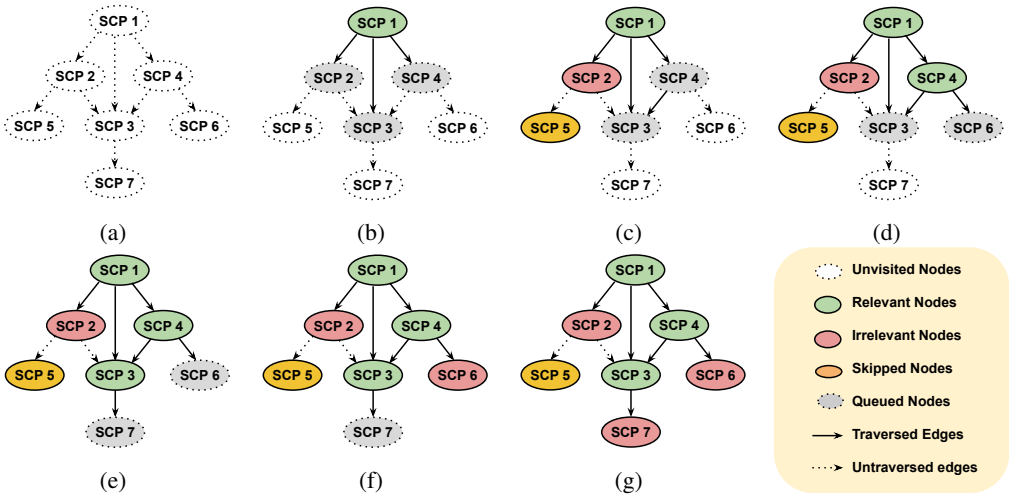
Step 2: Graph Traversal. With c_0, GRASP traverses G_{SCP} to incrementally refine the code c_i using relevant SCPs. Each node in the graph represents a distinct SCP, and edges encode dependencies that must be respected during traversal. Starting from the root, GRASP tracks two sets: $V_{visited}$ for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

Algorithm 1 Graph-Based Reasoning over SCPs

Require: scenario s, SCP Graph G_{SCP}, relevance threshold τ
Ensure: security-hardened code c_f

$c_0 \leftarrow \text{GenerateInitialCode}(s)$
$c_i \leftarrow c_0$; visited $\leftarrow \emptyset$; relevant $\leftarrow \emptyset$; stack $\leftarrow$ [root]
while stack $\neq \emptyset$ **do**
  $v \leftarrow$ stack.pop()
  if any parent of $v \notin$ visited **then**
    continue
  end if
  if parents of $v \neq \emptyset$ and all $\notin$ relevant **then**
    visited $\leftarrow$ visited $\cup\{v\}$
    continue
  end if
  $(R_i, c_i) \leftarrow \text{EvaluateAndUpdate}(v, c_i)$
  visited $\leftarrow$ visited $\cup\{v\}$
  if $R_i \geq \tau$ **then**
    relevant $\leftarrow$ relevant $\cup\{v\}$
    for each child in $G_{SCP}[v]$.children **do**
      if child $\notin$ visited **then**
        stack.push(child)
      end if
    end for
  end if
end while
$c_f \leftarrow \text{ReviseCode}(c_i)$
return c_f



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.

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

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. The prompt used for relevance assessment is provided in Appendix E.

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 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. The prompt used for this review is included in Appendix E.

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:

```
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.

```
 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.

```
     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.

```
    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. The final solution can be found in the Appendix B.

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 [23, 24, 42] focus on vulnerability-rich code but overlook whether functionality is preserved, since they lack unit tests for functional correctness. Correctness datasets [5, 29], such as HumanEval [9], 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. [16] 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 [44].

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` [52], and C scenarios are tested with compilation and shell scripts. Security analysis is conducted with CodeQL [11], following prior work [16, 23, 24], 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.1. Our benchmark consists of 54 natural language prompts (37 for Python and 17 for C) adapted from prior work [22, 23, 50], covering a total of 17 CWEs. The dataset is further detailed in Appendix A.2.

5.2 Experimental Setup

Models: We evaluate three proprietary models: OpenAI's GPT-4o-mini [45], Anthropic's Claude-3-haiku-20240307 [3], and Google's Gemini-1.5-flash-latest [20]; and one open-source model, Meta's Llama-3-8B-Instruct [37]. Generation for Llama 3 was conducted using three NVIDIA A40 GPUs [12]. 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 [23, 50], 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 [11] 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:

$$\text{SR} := \frac{\text{Secure and Valid samples}}{\text{Valid samples}}. \tag{1}$$

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. [16], we estimate secure-pass@k as:

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

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

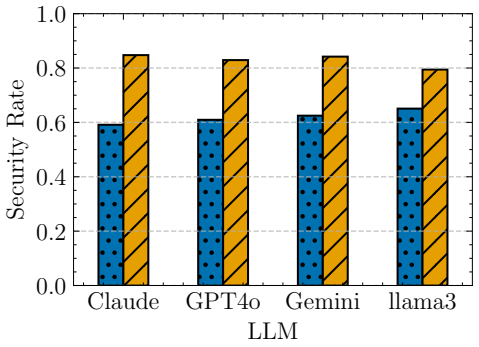


Figure 4: SR of Base Model and GRASP.

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**

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 C.2.

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 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) [59] strategy, and the most recent SOTA PromSec [42]. We do not include SVEN [23] or SafeCoder [24] 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. Nevertheless, since GRASP is model-agnostic, we also evaluate it against SVEN and SafeCoder on the smaller Phi-2 [28] model and present the results in Appendix C.1.

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 [59] builds on Chain-of-Thought [60] prompting and follows a structured multi-step reasoning process adapted from Ullah et al. [57]: (1) the model first plans the solution, (2) identifies vulnerable components, (3) analyzes these components for vulnerabilities, (4) considers

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**

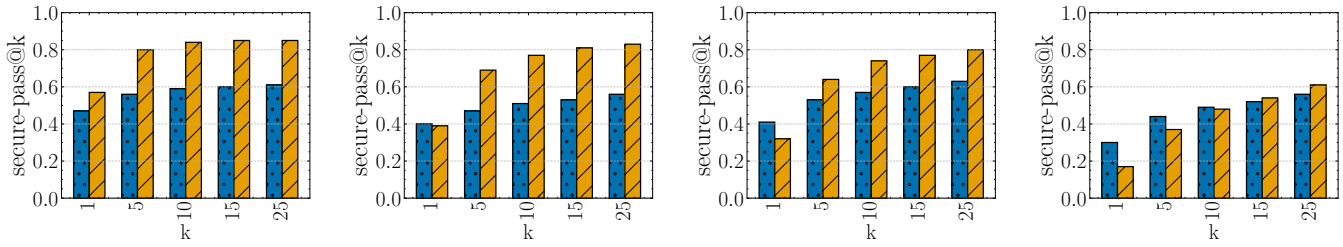


Figure 5: `secure-pass@k` scores for Base Model and GRASP.

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 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 [59], 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* [19], 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. The prompt used for this process is provided in Appendix E.

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 C.3. 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

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**

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 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

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

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

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

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 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 stronger security than baselines 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. Detailed results for Phi-2 can be found in Appendix C.1. These results underscore 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 [23], we use the state-of-the-art static analysis tool CodeQL [11] in our evaluations. To reduce false positives, we implement additional helper functions, again following the methodology of prior studies [23]. 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 [51] 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.

Ethical Considerations

This work involves the design, implementation, and evaluation of GRASP, a framework for fortifying large language models (LLMs) against generating insecure code. We considered multiple stakeholders impacted by our research: developers using LLM-based tools; end-users of software built with LLM-generated code; LLM providers whose models may be evaluated; and the broader security community.

The primary positive impact is the advancement of software security through systematic enforcement of Secure Coding Practices (SCPs) in LLM outputs, thereby reducing the risk of vulnerabilities in downstream applications. Our methodology does not involve human subjects or the collection of personal data, and thus presents no risks to privacy.

We identified potential harms, including that publishing vulnerability patterns could assist adversaries in crafting malicious prompts or code. There is also a reputational risk to LLM providers if vulnerabilities in their models are misinterpreted. To mitigate these risks, we ensured all demonstrations were generated in controlled environments using synthetic prompts. All examples were synthetic.

After weighing risks and benefits, we determined that the security improvements enabled by GRASP outweigh the minimal residual risks, given our mitigations. Our approach aligns with the Menlo Report principles: *Beneficence* by producing a strong net positive impact; *Respect for Persons* by avoiding human subject involvement; *Justice* through broad applicability without disproportionate harm; and *Respect for Law and Public Interest* through responsible, lawful conduct.

Open Science

In alignment with the USENIX Security '26 open science policy, we make available all artifacts required to reproduce and evaluate the contributions of this work in an anonymized repository. The repository includes: (1) the GRASP framework source code, covering both the SCP Graph Construction and Graph Reasoning modules; (2) the complete set of experimental scripts used for generating, evaluating, and analyzing LLM outputs; and (3) the full collection of prompts employed in our experiments. Each artifact is documented to support faithful replication of our results. The repository can be found in https://anonymous.4open.science/r/GRASP_USENIX_2026-3668.

No proprietary, sensitive, or personally identifiable data is contained in these artifacts. All datasets are synthetic and were generated specifically for this study. We do not provide direct access to commercial LLM APIs; instead, we supply scripts and instructions so reviewers can reproduce results with their own API credentials, ensuring adherence to provider terms of service.

Upon acceptance, we will release the non-anonymized repository under an open-source license for long-term availability. This ensures both immediate reproducibility for reviewers and broader community access, supporting transparency, replicability, and continued research into the security hardening of LLM-based code generation.

References

[1] Code generation on humaneval leaderboard. https://paperswithcode.com/sota/code-generation-on-humaneval. Accessed: 2024-09-01.

[2] Arafa Anis, Mohammad Zulkernine, Shahrear Iqbal, Clifford Liem, and Catherine 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)*, pages 618–625, 2018.

[3] Anthropic. Claude models documentation, 2024. Accessed: 2024-09-03.

[4] Owura Asare, Meiyappan Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code?, 2024.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[6] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, March 2024.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[8] CERT. Cert secure coding standards, n.d. Accessed: 2024-08-04.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[10] Claude. Claude. https://claude.ai/. Accessed: 2024-07-29.

[11] CodeQL. Codeql, 2024. Accessed: 2024-08-31.

[12] NVIDIA Corporation. Nvidia a40 datasheet, 2024. Accessed: 2024-09-05.

[13] DeepMind. Gemini, 2024. Accessed: 2024-08-31.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2023.

[16] Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. Constrained decoding for secure code generation, 2024.

[17] GitHub. Github copilot, 2024. Accessed: 2024-08-04.

[18] GitHub. Research: Quantifying github copilot's impact on developer productivity and happiness, 2024. Accessed: 2024-08-04.

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

[20] Google DeepMind. Gemini flash models documentation, 2024. Accessed: 2024-09-03.

[21] Google Research. Ml-enhanced code completion improves developer productivity, 2024. Accessed: 2024-08-31.

[22] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario 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)*, pages 684–709. IEEE, 2024.

[23] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23. ACM, November 2023.

[24] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation, 2024.

[25] IBM. Ibm data breach reports, 2024. Accessed: 2024-08-31.

[26] IBM. Reputational it risk report, 2024. Accessed: 2024-08-31.

[27] Invicti. Invicti, 2024. Accessed: 2024-08-31.

[28] Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, Suriya Gunasekar, Piero Kauffmann, Yin Tat Lee, Yuanzhi Li, Anh Nguyen, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Michael Santacroce, Harkirat Singh Behl, Adam Taumann Kalai, Xin Wang, Rachel Ward, Philipp Witte, Cyril Zhang, and Yi Zhang. Phi-2: The surprising power of small language models. *Microsoft Research Blog*, 2023.

[29] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[30] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt?, 2023.

[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muh-

tasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[33] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156, 2012.

[34] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 372–383, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Na Meng, Stefan Nagy, Daphne Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: Challenges and vulnerabilities, 2017.

[36] Meta. Llama, 2024. Accessed: 2024-08-31.

[37] Meta. Meta llama 3 8b model documentation, 2024. Accessed: 2024-09-03.

[38] Microsoft. Secure coding guidelines, 2022. Accessed: 2024-08-04.

[39] MITRE. Cwe top 25 most dangerous software errors: 2023 edition, 2023. Accessed: 2024-08-31.

[40] MITRE. Common weakness enumeration (cwe), 2024. Accessed: 2024-08-31.

[41] Yutao Mou, Xiao Deng, Yuxiao Luo, Shikun Zhang, and Wei Ye. Can you really trust code copilots? evaluating large language models from a code security perspective, 2025.

[42] Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai 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*, pages 2266–2280, 2024.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[44] OpenAI. Openai. https://platform.openai.com/docs/models. Accessed: 2024-07-29.

[45] OpenAI. Openai api models documentation, 2024. Accessed: 2024-09-03.

[46] OpenAI. Openai codex, 2024. Accessed: 2024-08-31.

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

[48] OWASP. Owasp zap, 2024. Accessed: 2024-08-31.

[49] OWASP Foundation. Owasp secure coding practices, n.d. Accessed: 2024-08-04.

[50] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions, 2021.

[51] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey, 2024.

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

[53] Jakub Res, Ivan Homoliak, Martin Perešíni, Aleš Smrčka, Kamil Malinka, and Petr Hanacek. Enhancing security of ai-based code synthesis with github copilot via cheap and efficient prompt-engineering, 2024.

[54] Semgrep. Semgrep, 2024. Accessed: 2024-08-31.

[55] SonarSource. Sonarqube, 2024. Accessed: 2024-08-31.

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

[57] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks, 2024.

[58] Veracode. Dynamic analysis (dast), 2024. Accessed: 2024-08-31.

[59] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, 2023.

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[61] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

A Dataset Details

A.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
```

Table 8: Details of Benchmark Dataset Distribution.

CWE	Description	# py	# C	Task Types Covered
CWE-020	Input validation	3	0	Unvalidated redirects, domain validation, untrusted file input (YAML)
CWE-022	Path traversal	5	1	Path traversal in file access, Flask file serving, file deletion, tar extraction
CWE-078	OS command injection	5	2	OS command execution, user input in shell commands, subprocess usage
CWE-089	SQL injection	2	0	SQL queries with user input, email lookup, stock order insertion
CWE-094	Code injection	1	0	Dynamic import via exec, version-based module loading
CWE-125	Out-of-bounds read	0	3	Array index from user input, out-of-bounds checks, C array access
CWE-190	Integer overflow	0	3	Integer addition, large constant addition, overflow from user/random input
CWE-215	Debug info exposure	1	0	Intentional crash, Flask exception, debug exposure
CWE-416	Use after free	0	2	Manual memory management, buffer allocation and access, potential use-after-free
CWE-476	NULL pointer dereference	0	2	Dynamic memory allocation, struct pointer usage, NULL safety checks
CWE-502	Insecure deserialization	6	0	Untrusted deserialization, pickle, yaml input, base64 decoding, Flask route parsing
CWE-601	Open redirect	4	0	Open redirect, URL validation, host allowlist, Flask/Django redirects
CWE-611	Improper XML Restrictions	5	0	untrusted XML input, file-based parsing
CWE-643	XPath injection	2	0	Unvalidated user input in XPath queries, XML search by attribute
CWE-676	Dangerous library functions	0	3	Unsafe input handling, use of standard library I/O, time-based logic
CWE-732	Improper file permissions	1	1	File permission restrictions, world-writable file prevention, access control
CWE-918	SSRF	2	0	External URL fetch, user-controlled input in HTTP requests, SSRF risk

A.2 Dataset Scenarios

Table 8 summarizes our benchmark, which spans a broad range of CWEs commonly observed in real-world systems. For each CWE, we curated a set of prompt scenarios that are intended to elicit security-sensitive behaviors from LLMs. Each scenario corresponds to a case where code may be insecure if not handled properly, thereby allowing us to evaluate whether models recognize the risk and apply the appropriate defensive strategies. To broaden coverage, the dataset includes both Python and C scenarios, enabling us to examine whether security-aware code generation generalizes across programming languages rather than being confined to a single ecosystem.

Table 8 also enumerates the number of scenarios associated with each CWE and provides short descriptions of the security-relevant behaviors or tasks that the prompts are designed to test. For example, CWE-022 (Path Traversal) includes prompts involving insecure file access, file deletion, and tar archive extraction—each one crafted to test whether the model correctly implements directory traversal defenses. By grounding each scenario in a concrete and recognizable programming task, we ensure that the evaluation remains both realistic for practitioners and targeted to specific classes of vulnerabilities.

By prompting models with these scenarios, we directly evaluate whether they apply SCPs such as input validation, output encoding, and file path sanitization, providing a systematic assessment of how well LLMs reason about security requirements across diverse vulnerabilities and application contexts. In doing so, the benchmark highlights both the strengths of current models and the areas where they continue to fall short.

B Full End-to-End Example

The example below shows the final output produced by GRASP after completing a full traversal of the SCP Graph for the task outlined in Section 4.4. As discussed in Section 4, GRASP applies Secure Coding Practices (SCPs) in a dependency-respecting manner, selectively invoking only those practices that are relevant to the scenario.

In this case, the original task is to extract a compressed tar archive to a specified directory. GRASP incrementally refines the code by applying three SCPs. First, SCP 1 is applied to ensure that the extraction directory exists and is valid. Next, SCP 3 is used to prevent directory traversal attacks by confirming that the resolved extraction paths remain within the target directory. Finally, SCP 4 is applied to prevent overwriting existing files during extraction. The resulting implementation below reflects a secure and functionally correct solution, illustrating the output of GRASP's structured reasoning process.

```
import tarfile
import os
def extract():
    archive_file = "archive.tar.gz"
    extract_dir = "/tmp/unpack"
    # Ensure the extraction directory exists and
        is a directory (SCP 1)
    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")
    with tarfile.open(archive_file, 'r:gz') as tar
        :
        for member in tar.getmembers():
            target_path = os.path.join(extract_dir
                , member.name)
```

```
            # Prevent directory traversal (SCP 3)
            if not os.path.commonpath((extract_dir
                , target_path)) == extract_dir:
                raise ValueError("Extraction path
                    is outside of the target
                    directory or contains ..
                    element")
            # Prevent overwriting existing files (
                SCP 4)
            if os.path.exists(target_path):
                raise ValueError(f"File '{member.
                    name}' already exists in the
                    extraction directory")
            tar.extract(member, path=extract_dir)
extract()
```

C Evaluation Details

C.1 Additional Comparison on Small-Scale Models

To complement our main experiments on modern instruction-following LLMs such as GPT-4o, Claude, Gemini and Llama, we additionally evaluate GRASP against smaller open-source baselines aligned with prior tuning-based approaches. In particular, we include comparisons with SVEN [23] and SafeCoder [24], both of which were originally proposed for lightweight models with limited in-context reasoning capacity. For reference, we also report results from a Base model without specialized tuning. Unlike SVEN and SafeCoder, which require model-specific fine-tuning, GRASP is applied directly via graph reasoning without any additional training.

Table C.1 reports performance across multiple metrics, including valid unique generations, secure generations (sec), Security Rate (SR), and secure-pass@k for $k \in 1, 5, 10, 15, 25$. As expected, overall scores are lower than those obtained with large instruction-tuned LLMs. SafeCoder achieves the highest SR, but this comes at the expense of functional correctness, producing many syntactically invalid or functionally incorrect outputs. SVEN, meanwhile, fails entirely in this setting. Both methods are highly dependent on the datasets they were trained on, and without exposure to similar vulnerabilities during training, they struggle to generalize and cannot reliably generate secure and correct code. GRASP, by contrast, performs similarly to the Base model, since smaller models do not have the reasoning capacity to benefit from SCP-guided refinements.

These results emphasize the different scopes of these approaches. SVEN and SafeCoder were designed for smaller models and may be more appropriate in that regime, whereas GRASP targets larger instruction-tuned LLMs, where it scales effectively without requiring task-specific tuning or additional training. Our main evaluations confirm that, within this intended scope, GRASP consistently strengthens security while preserving functional correctness.

Table 9: GRASP vs. Baselines with GPT4o.

Model	Val	Sec	SR	secure-pass@k				
				1	5	10	15	25
Base	1514	1114	0.74	0.14	0.29	0.37	0.43	0.48
Sven	1023	689	0.67	0.12	0.22	0.25	0.27	0.30
Safecoder	731	647	0.88	0.02	0.05	0.06	0.07	0.07
GRASP	1090	748	0.69	0.09	0.23	0.31	0.35	0.41

C.2 Detailed Results of CWE specific performance

Tables 10 and 11 report results for CWEs not included in the main breakdown of Table 2. In several of these scenarios, the Base model already achieves near-perfect or perfect Security Rates (SR), such as for CWE-089 (SQL Injection), CWE-094 (Code Injection), and CWE-416 (Use After Free). These categories remain highly critical in practice and are regularly highlighted in industry security benchmarks. GRASP achieves comparable results in these cases, demonstrating that it preserves the strengths of the Base model and does not degrade performance where models already perform well. At the same time, GRASP still provides room for notable improvements. For instance, on CWE-094, Claude improves from an SR of 0.21 to 1.00 showing GRASP's ability to enforce secure practices often missed in base completions.

Beyond these widely prevalent categories, GRASP also demonstrates substantial improvements on CWEs that are less frequently evaluated yet equally important. For example, in CWE-502 (Insecure Deserialization), Gemini's SR improves from 0.33 to 0.74, and Claude improves from 0.33 to 0.69. These vulnerabilities often demand reasoning about object state and serialization logic—areas where base models commonly falter. Likewise, in CWE-601 (Open Redirect) and CWE-611 (XXE), GRASP raises SR across multiple LLMs, showing that SCP-guided refinement generalizes effectively to input/output processing flaws as well. In especially difficult cases such as CWE-643 (XPath Injection), where the Base Model nearly fails completely, such as Claude's SR of 0.02, GRASP still lifts performance dramatically (0.72), demonstrating that even highly specialized vulnerabilities can benefit from structured reasoning guidance.

Finally, in more moderate categories like CWE-732 (Incorrect Permission Assignment), GRASP consistently maintains or improves SR across the board, reinforcing that its benefits extend beyond narrow or extreme cases. While a few minor regressions exist (e.g., GPT-4o on CWE-676), these are small, rare, and far outweighed by the overall trend. Taken together, these results highlight GRASP's robustness across vulnerabilities, and further indicate that SCP-guided reasoning scales reliably across domains and security patterns, providing consistent improvements without introducing new weaknesses.

Table 10: 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 11: 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

C.3 Additional Results of Zero Day Vulnerability Evaluations

Tables 12, 13 and 14 present detailed results for a broader set of real-world CVEs beyond those covered in Section 5.6. These evaluations further illustrate how GRASP performs across diverse zero-day settings, where models are prompted to reproduce vulnerable implementations.

In some CVEs such as `CVE-2023-45670`, all methods, including the Base model, achieve perfect security rates. Even though the prompts explicitly instruct the model to produce vulnerable code, the Base model avoids generating insecure implementations in these cases. This suggests that certain vulnerabilities are straightforward for models to handle, and importantly, it shows that GRASP preserves strong baseline performance rather than introducing regressions.

The contrast is sharper in CVEs where baseline methods fail almost entirely. For example, in `CVE-2023-50264`, GRASP improves the SR from 0 to 0.8, while in `CVE-2024-31451` and `CVE-2024-32022`, it raises SRs from 0 to 0.88 and 0.64, respectively. Strikingly, for `CVE-2024-32025`, GRASP achieves a perfect SR of 1.0 while all baselines remain near zero. These gains highlight GRASP's ability to guide models toward secure completions even in highly adversarial prompts.

Not all cases yield such dramatic improvements. In CVEs such as `CVE-2025-27783` and `CVE-2025-27784`, GRASP achieves only modest security rates, though still comparable to or better than alternatives. These results show that while challenges remain, GRASP rarely performs worse than the baselines and often delivers meaningful improvements.

Overall, these additional results reinforce the central finding: GRASP generalizes effectively to a wide range of zero-day vulnerabilities. By maintaining performance in cases where models already succeed and substantially improving outcomes in harder scenarios, it demonstrates robustness and adaptability without relying on retraining or dataset-specific tuning.

Table 12: 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**

Table 13: 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 14: Security Rate for Zero Day Evaluation

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

D SCP Graph Construction

The construction of the SCP Graph proceeds in several stages, as outlined in Section 4.2.3 of the main paper. Here, we elaborate on the details of the approach, formalized in Algorithm 2. The prompts for each step is included in Appendix E.

We begin with a set of SCPs S, drawn from the OWASP Secure Coding Practices Checklist and filtered to retain only code-level practices that mitigate MITRE Top 25 CWEs [39]. Each SCP is first *normalized* into a structured JSON format using the LLM, preserving its description, scope, actions, examples, and conditions. This ensures consistency across SCPs and provides machine-readable fields for downstream reasoning.

Next, the algorithm performs *pairwise relation classification*. For each ordered pair of SCPs (s_i, s_j), the LLM determines whether a direct dependency exists, and if so, whether it is a Sequential dependency or a Specificity dependency, as described in Section 4.2.2. If no direct relationship exists, no edge is added.

From this set of dependencies, we construct an initial graph. Since classification may introduce cycles, the next step is *cycle elimination*. Whenever a directed cycle is detected, the LLM is consulted to decide which edge should be removed while preserving semantic correctness. This guarantees that the resulting SCP Graph is a *directed acyclic graph (DAG)*.

The graph is then refined by removing *redundant edges*. In cases where a shortcut edge exists (e.g., $A \to C$) alongside an intermediate path ($A \to B \to C$), the LLM is asked whether C can ever be directly applied after A without requiring B. If not, the shortcut edge is removed. This prevents spurious dependencies while retaining edges necessary for SCPs that may apply independently of intermediates.

Finally, the graph is *rooted* by introducing a synthetic "root" node connected to all source nodes (i.e., those without incoming edges). This guarantees that all SCPs are reachable, including practices that do not depend on specific predecessors.

Because the graph is constructed with LLM assistance, we include a manual verification step to ensure correctness. In this paper, we primarily used GPT-4o-mini to generate the graph, though experiments with other LLMs produced largely similar structures. Manual verification revealed that approximately 60–70% of the original edges were the same before and after manual verification. During the manual verification process, we added edges that the LLM had omitted due to its limited global context and removed edges that, while not strictly incorrect, were redundant.

Algorithm 2 Automated Construction of the SCP Graph

Require: SCPs S, LLM M
Ensure: SCP Graph G_{SCP}

$S' \leftarrow \text{Normalize}(S, M)$
$E \leftarrow \emptyset$
for each $(s_i, s_j) \in S' \times S'$ **do**
  $r \leftarrow \text{ClassifyRelation}(s_i, s_j, M)$
  if $r \in \{\text{SEQUENTIAL}, \text{SPECIFICITY}\}$ **then**
    $E \leftarrow E \cup \{(s_i, s_j, r)\}$
  end if
end for
$G \leftarrow (S', E)$
while G has cycle **do**
  $e \leftarrow \text{SelectEdgeToRemove}(M, \text{cycle})$
  $E \leftarrow E \setminus \{e\}$
end while
for each edge $(a, c) \in E$ **do**
  if $\exists b : (a, b), (b, c) \in E$ and $\text{IsRedundant}(a, b, c, M)$ **then**
    $E \leftarrow E \setminus \{(a, c)\}$
  end if
end for
$G \leftarrow (S' \cup \{\text{root}\}, E \cup \{(\text{root}, v) : v \in \text{Sources}(G)\})$
return G

Table 15: 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.	

E Prompt Templates

Prompt template for structured extraction of Secure Coding Practices (SCPs).

You are given a Secure Coding Practice (SCP) or policy statement. Extract information to fill out the following structured format, using only content that is explicitly present in the provided text. If a field is not explicitly stated in the input, leave it blank (`""` or `[]` as appropriate). The `practice` field must include the SCP exactly as written in the input. Your output should be in the following JSON format:

```
{
  "practice": "",
  "category": "",
  "coding_phase": "",
  "application_scope": "",
  "examples": [],
  "key_terms": [],
  "notes": ""
}
```

Field Instructions:

- **practice**: Use the SCP or mitigation as stated in the text. This can never be blank. Just copy the input.
- **category**: Use an explicit category from the text if available (e.g., Input Validation); otherwise, leave blank.
- **coding_phase**: Use the explicit development phase stated in the text (e.g., Implementation), or leave blank.
- **application_scope**: Use only the application scope as stated in the text.
- **examples**: List only examples explicitly stated in the text.
- **key_terms**: List only key terms or technical concepts explicitly stated in the text.
- **notes**: Use only if explicit annotator comments or notes are present; otherwise leave blank.

Secure Coding Practice: `{scp_text}`

Prompt template for extracting dependencies between Secure Coding Practices (SCPs).

You are provided with two Secure Coding Practices (SCPs). Your task is to determine if there is a meaningful, direct dependency between them. Your goal is to identify relationships that a security-aware developer would rely on when implementing secure systems.
There are three types of relationships:

1. **Sequential Dependency:** One SCP must be applied before the other to ensure security. This includes cases where the first SCP is a defensive safeguard that enables the safe application of the second SCP. Examples include input validation or sanitization before dangerous operations such as deserialization, OS commands, redirects, or user-input logging.
2. **Specificity Dependency:** One SCP is a more specific implementation, technique, or refinement of a broader SCP. This includes cases where the second SCP is a concrete example or detailed approach under the same context, such as file validation, output encoding, or input sanitization.
3. **None of the Above:** The SCPs are not meaningfully dependent in terms of order or specificity. They may share a security domain but do not directly influence or refine one another.

Examples: `{examples}`
Guidelines:

- Prioritize identifying valid dependencies that would help reduce the number of missing edges in a secure coding graph.
- Consider the relationship between the SCPs in isolation, based on practical implementation logic.
- **Direction matters:**
  Sequential: the prerequisite comes first.
  Specificity: the broader principle comes first.
- Only use "None of the above" if there is truly no meaningful conceptual or operational link. Prefer to identify a valid dependency if there is reasonable justification.

Output Format: `{output_format}`
Now analyze the following pair: `{scp_a_json}` `{scp_b_json}`

Prompt template for deciding whether to keep or remove direct edges implied by longer SCP paths.

You are given a set of Secure Coding Practices (SCPs), each described in JSON. There may be two types of relationships between these SCPs:

- **Sequential Dependency**: One SCP must be applied before the other for secure coding to be effective.
- **Specificity Dependency**: One SCP is a more specific or detailed case of a broader SCP.

SCPs:
`{scp_definitions}`
Existing relationships in the current graph:
`{relationship_descriptions}`
There is a direct dependency from `{ancestor_id}` to `{descendant_id}`, but also a longer path:

`{ancestor}` to `{intermediates}` to `{descendant}`

Your task: Decide whether the direct edge (`{ancestor_id}` → `{descendant_id}`) should be *kept* or *removed*.
Decision Rules:

- If *any* relationship on the intermediate path is *Sequential*, remove the direct edge unless a concrete, real workflow safely skips *all* intermediates without weakening security.
- If *all* relationships are *Specificity*, remove the direct edge unless the intermediates are fully redundant in practice.
- If there is any doubt, remove the edge.
- Do not keep the edge based on abstract logic; only keep it if a practical workflow justifies skipping all intermediates without loss of security intent.

Examples (parameterized):
`{examples}`

Output Format (parameterized):
`{output_format}`

Instance to analyze:
Ancestor: `{ancestor}`
Descendant: `{descendant}`
Intermediates: `{intermediate_chain}`

Prompt template for resolving cycles in an SCP dependency graph.

You are given a directed cycle in a Secure Coding Practice (SCP) dependency graph. Choose exactly *one* directed edge to remove in order to break the cycle while preserving meaningful security semantics.
Guidelines:

- Prefer removing weaker or redundant edges.
- Prefer removing direct edges that merely duplicate longer dependency chains.
- Justify the removal with concrete reasoning tied to the practices involved.

Examples (parameterized):
`{examples}`

Output Format (parameterized):
`{output_format}`

Instance to analyze:
Cycle: `{cycle}`

Prompt template to (a) assess SCP relevance and (b) apply the SCP if deemed applicable.

Prompt: `{scenario}`
Code: `{current code}`
You are given a prompt scenario and a corresponding code snippet above. Your objective is to assess the applicability and relevance of the Secure Coding Practice (SCP) `"{practice}"` to the provided code and scenario.
1. Explain your reasoning clearly—discuss whether and how the SCP is applicable to the current code. Consider both the scenario and the code behavior.
2. Analyze the code and its dataflow and then rate the relevance of the SCP using the following scale:
Relevance Levels:
- 1 - Not Relevant: The SCP is entirely unrelated to the code or scenario. It does not apply in this context.
- 2 - Slightly Relevant: The SCP might apply under different circumstances, but has very limited applicability here.
- 3 - Moderately Relevant: The SCP applies to some extent, and its application may provide moderate improvement or risk reduction.
- 4 - Relevant: The SCP is clearly applicable and its implementation will improve the security posture of the code.
- 5 - Very Relevant: The SCP directly addresses a core security concern in the code and is strongly recommended for implementation.
3. If the SCP's relevance is at least `{relevance_threshold}` and it is not already implemented, integrate it directly within the existing function(s) in the code. Do not create separate functions solely for implementing the SCP. Ensure:
- The original functionality of the code is preserved (functional correctness must not be compromised).
- Security improvements are inline and context-aware.
If there are changes in the code, ensure that the original prompt is still addressed and the code remains functionally correct. Note that the original prompt may not include the security features implemented. The security features implemented should not be reverted. Your output should contain the entire code in its entirety, including all original headers, imports, unchanged code, and the updated implementation. Do not revert or remove previously implemented SCPs. Do not create new functions. If you implement any SCP make sure the code for it is within the existing functions itself.
If no changes are needed, clearly indicate that under the "Entire Code" section.
Your output should be in the following format:
Explanation: `[Your explanation here]`
Relevance: `[1-5]`
Entire Code: `[The new code in its entirety if changes were implemented]` OR `[The text "No Changes" if no changes were implemented]`

Prompt used for Functional Correctness verification

Prompt: `{scenario}`
Secure Code: `{current_code}`
The code above shows a secure implementation of the given prompt. Your task is to verify that the security measures do not compromise functional correctness.
Please follow the steps below:
1. Understand the original prompt and identify the core functionality it is meant to implement.
2. Check the code against that functionality step-by-step. Does it do exactly what was asked, or are there logic errors or deviations?
3. Verify the behavior using test cases:
- If test cases are already included in the prompt, ensure that:
• They are correctly handled by the implementation
• The expected outputs and status codes (if applicable) are accurate
- If no test cases are provided, construct your own test cases to cover:
• Empty or missing input (e.g., empty strings, empty lists, `None`)
• Invalid inputs (e.g., disallowed characters, out-of-bound values)
• Valid typical input
• Valid edge and boundary cases
4. Ensure all security measures remain in place. Do not remove any security measure unless:
- It directly interferes with the required functional behavior
- You are replacing it with a functionally correct and equally secure alternative
5. Fix any bugs or edge case failures without compromising the security model.
6. If changes were necessary, provide the full corrected version of the code, including all imports, definitions, and unchanged parts. Your output should be in the following format:
Explanation: `[Your explanation here]`
Relevance: `[1-5]`
Entire Code: `[The new code in its entirety if changes were implemented]` OR `[The text "No Changes" if no changes were implemented]`

Prompt template for lossless natural-language reproduction of Python code.

Estimate a highly detailed natural language prompt that could be given to an LLM to generate the following Python code.
The goal is for the code produced from this prompt to be **exactly identical** to the input code — line-for-line, character-for-character. The prompt must be a **lossless natural language encoding** of the original code. It is not a summary or rephrasing. All code structure, syntax, formatting, comments, and ordering must be **perfectly preserved**.
Instructions:

- The prompt must describe the code so literally and precisely that an LLM can reconstruct it **exactly**, with no differences.
- Do **NOT** interpret, summarize, simplify, generalize, rename, reorder, paraphrase, optimize, or omit anything.
- The prompt must be written entirely in **natural language** — it must not contain code blocks or inline code.
- You must describe **every line**, including:
  - All **import statements** (with aliases and exact order)
  - All **global variables or constants**, even if undefined
  - All **function or class definitions** (with full parameters, decorators, and logic)
  - Every **loop, conditional, list or dict operation, method call, expression**, and **return**
  - Every **string literal**, whether f-string, formatted, or raw
  - Every **comment**, in its **exact wording, location, indentation**, and **format** (inline or standalone)
- When subprocess or shell commands are constructed (e.g., command lists), describe **every argument** in **the correct order**, exactly as written, including flags and values.
- If the code references undefined constants or globals (like `CACHE_DIR`), **mention them by name** in the prompt and preserve them in place.

Any deviation — such as dropped comments, reordered expressions, shortened strings, or cleaned-up formatting — is a failure.
Your output should follow this format:
`{Output Format}`
Example: `{Example}`
Now process the following code accordingly:
The Python Code is given below:
`{python_code}`