GUIDED REASONING WITH ACTIONABLE SECURITY PRINCIPLES FOR SECURE CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models generate code that is frequently insecure, and existing defenses are *vulnerability-centric*: they teach models what not to do, by fine-tuning on vulnerable and fixed code pairs, by retrieving records of past vulnerabilities, or by revising drafts under static analyzer feedback. Such defenses depend on curated vulnerability data, generalize poorly to weaknesses absent from it, and are often impractical for proprietary models. We take a *principle-centric* direction instead: rather than describing past vulnerabilities, we condition generation on prescriptive Secure Coding Principles that state how secure code should be written. We present GRASP, which organizes 214 such principles into a directed acyclic graph capturing specialization and prerequisite dependencies, and traverses it at inference time, selecting the principles relevant to a task and applying them in dependency order while pruning branches that do not arise. GRASP requires no training, no weight access and no analyzer in the generation loop, so it applies to proprietary and open-weight models alike. Across five models we raise the Security Rate from 0.46–0.55 at baseline to 0.70–0.84, outperform zero-shot, planning-based, retrieval-augmented and analyzer-guided prompting on secure-pass@k on our dataset, and improve robustness on unseen vulnerabilities, measured on real CVEs disclosed after the evaluated models' training cutoffs.

1 INTRODUCTION

Large language models (LLMs) are increasingly used to generate code and assist software development (OpenAI, 2024a; Anthropic, 2024b; DeepMind, 2024), with reported gains in developer productivity (Kalliamvakou, 2024; Google Research, 2024). However, these gains do not ensure the security of generated code. Prior studies have identified vulnerabilities in LLM-generated programs (Pearce et al., 2021; Asare et al., 2024). For example, Pearce et al. (2021) found that approximately 40% of Copilot-generated programs were vulnerable in their evaluation, and Mou et al. (2025) report secure-generation rates of 70–75% for models such as GPT-4o on their evaluated tasks. These findings highlight the need for code-generation methods that address security requirements alongside functional correctness.

A preliminary study further illustrates the limitations of a general security instruction. On a deliberately challenging 20-scenario Python subset covering path traversal, command injection, insecure deserialization, open redirects, XML and XPath injection, XSS, information exposure, and SSRF, the five LLMs we evaluate achieve Security Rates ranging from 0.22 to 0.36. Adding an explicit security instruction to the prompt raises the rate to at most 0.58, but a substantial fraction of the generated code remains vulnerable. Instructing a model to be security-aware is therefore not sufficient, which motivates guiding it through structured reasoning over the obligations that govern the task rather than through a general exhortation to be careful. Appendix A.1 reports the study in full.

Existing defenses adopt a *vulnerability-centric* paradigm, teaching models "what not to do" either by training on vulnerable and fixed code pairs (He & Vechev, 2023; He et al., 2024; Liu et al., 2025; Xu et al., 2025) or by leveraging external analyzers to suppress known insecure patterns (Nazzal et al., 2024). Section 2 treats each family in turn. While effective against certain vulnerabilities, they share persistent limitations: reliance on large curated datasets, poor generalization to unseen weaknesses, and limited practicality for proprietary models, which Appendix A.2 sets out in full. Collectively

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

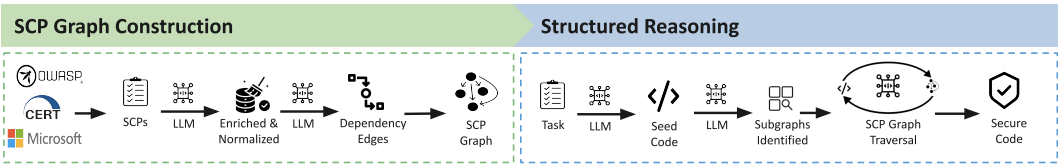

Figure 1: Overview of GRASP. Graph construction runs once, offline, to organize secure coding principles by their dependencies; structured reasoning runs per task, to select the principles relevant to it and apply them in order.

these underscore the *reactive* nature of existing methods, which start from observed vulnerability patterns rather than from the security principles that should govern correct code behavior.

Human developers avoid writing vulnerable code differently. They follow well-established Secure Coding Principles (SCPs), guidelines that help them *proactively* reason about risk before vulnerabilities manifest (OWASP Foundation, n.d.; CERT, n.d.; Microsoft, 2022). Given that LLMs demonstrate strong reasoning and comprehension capabilities (Brown et al., 2020; Chang et al., 2024), we investigate whether they can be guided to follow structured, principle-driven development. This shifts the focus from teaching models "what not to do" toward teaching them "what to do" and "how to do it". We call this a *principle-centric* paradigm, one that emphasizes proactive reasoning rather than reactive avoidance.

We introduce GRASP (**G**uided **R**easoning with **A**ctionable **S**ecurity **P**rinciples), shown in Figure 1, which operationalizes this paradigm through two components: an *SCP Graph* that organizes security principles into a directed acyclic graph encoding their dependencies, and a *structured reasoning* process that guides LLMs to identify, interpret and apply task-relevant SCPs during generation. Instead of retraining models on vulnerability datasets or relying on external analyzers at inference, GRASP embeds SCPs directly into the model's reasoning process, yielding adaptability to unseen vulnerabilities, model-agnostic integration and interpretable security decisions.

In summary, our contributions are fourfold. (1) We investigate Secure Coding Principles as a foundation for proactive LLM-based secure code generation, showing how prescriptive obligations can guide models beyond vulnerability-pattern avoidance. (2) We propose GRASP[1], which organizes 214 principles into a directed acyclic graph capturing specialization and prerequisite dependencies and traverses it at inference time. (3) We curate an evaluation set of 54 natural-language prompts spanning 17 CWEs in Python and C, in which every task carries both unit tests and vulnerability checks, and we additionally evaluate on CWEval and BaxBench so that the results do not rest on data of our own construction. (4) We evaluate across five LLMs, three benchmarks, five prompting, retrieval and analyzer-guided baselines, two fine-tuned checkpoints and 20 real CVEs, and show through paired ablations which components contribute to the observed gains.

2 BACKGROUND AND RELATED WORK

Before describing GRASP, we set out what existing defenses condition on, since that is what separates them from a principle-centric approach, and what secure coding principles are.

Secure code generation. Existing work follows a vulnerability-centric paradigm and divides into three families. *Tuning-based* methods modify parameters: SVEN (He & Vechev, 2023) applies prefix tuning over vulnerable and fixed pairs, SafeCoder (He et al., 2024) performs security-focused instruction tuning on verified fixes mined from commits, PurpCode (Liu et al., 2025) combines security reasoning with reinforcement learning, and ProSec (Xu et al., 2025) synthesizes CWE-based scenarios with preference learning. These require weight access, ruling out proprietary models, and depend on curated vulnerability data. *Retrieval-based* methods keep weights fixed and condition generation on an external corpus of security knowledge, applying retrieval-augmented generation (Lewis et al., 2021) to the security setting, as SecCoder (Zhang et al., 2024) does by retrieving

[1]Code and data: `https://anonymous.4open.science/r/GRASP_ICLR-09C7`.

safe demonstrations for in-context learning. RESCUE (Shi & Zhang, 2026) distills a hierarchical knowledge base from vulnerability-fix pairs and retrieves guidelines and sliced code examples from it. Our RAG baseline in Section 4.1 instantiates the simpler version of this family. *Analyzer-guided* methods use static analysis feedback to drive refinement, as PromSec (Nazzal et al., 2024) does, and INDICT (Le et al., 2024) pairs a safety critic with a helpfulness critic so that the two objectives are traded off explicitly rather than implicitly. Unlike all three, which teach a model what insecure patterns to avoid, GRASP guides it through structured reasoning over principles that state what secure code should do and how to implement it.

Secure coding principles. SCPs specify the obligations code should satisfy to avoid insecure behavior (Meng et al., 2017). OWASP Secure Coding Practices (OWASP Foundation, n.d.), CERT Secure Coding Standards (CERT, n.d.) and vendor guidelines (Microsoft, 2022) cover input validation, encoding, and session management. Because SCPs describe how secure implementations should be constructed rather than what past attacks looked like, they are a natural foundation for proactive generation. Appendix K gives more details.

3 GRASP

Operationalizing secure coding principles is not a matter of putting a checklist in the prompt. Published guidelines (OWASP Foundation, n.d.; CERT, n.d.) contain hundreds of principles, many context-specific or interdependent. Presenting all of them at once risks exceeding context windows, applying irrelevant guidance, and producing redundant or incorrectly ordered refinements. A flat checklist also treats principles as independent instructions, when in fact some specialize, depend on, or enable others. GRASP addresses this with the two components illustrated in Figure 1: an SCP Graph that captures prerequisite and specialization relationships, and a structured reasoning process that uses the graph to select task-relevant principles, apply them in a dependency-aware order, and prune irrelevant branches during iterative refinement.

3.1 SCP GRAPH

Representation. Each SCP is a self-contained natural-language node describing a code-level security obligation, which is what makes the principles *actionable*: each one names something a program can be made to do. We focus on principles that can directly guide generation, such as validating file paths before access, using parameterized queries for database operations, checking numeric ranges before arithmetic, or avoiding exposure of sensitive information in error messages. This code-level focus distinguishes SCPs from organizational or deployment guidance such as network segmentation, which matters for security but cannot be enforced by modifying generated code. Natural language is used deliberately, so that principles can be interpreted by any model at inference time without model-specific training or weight access.

Relationships. Edges capture two relations. *Specialization* connects a broad principle to a more concrete obligation that refines it, for example from a general requirement to prevent injection to a specific requirement to use parameterized queries, allowing GRASP to reason from high-level goals toward actionable requirements. *Prerequisite* captures ordering constraints: input validation should be considered before the error handling that reports validation failures, and validating a file path should precede operations that open or overwrite files. At inference both relations act as parent constraints in the same way: a principle is considered only once every parent of either kind has been considered and at least one of them has been applied.

Construction. We build the graph from the OWASP Secure Coding Practices checklist (OWASP Foundation, n.d.), extracting 214 principles and preserving their association with the checklist's 14 categories. Construction is automated and proceeds in four stages. First, each extracted principle is normalized and enriched into a structured description clarifying the obligation and adding representative insecure and secure usage patterns; this stage runs three LLMs independently over every principle and combines their outputs. Second, a single model judges all 22,791 unordered principle pairs once each, classifying the relation as specialization, prerequisite, or none, which yields 888 edges. Third, because the raw result is heavily cyclic, a model from a different family reviews the 406 structurally suspect pairs, namely those participating in cycles or reciprocal edges, and prunes 27% of the inferred edges, reducing the largest strongly connected component from 83 nodes to a

single reciprocal pair, which is dropped rather than ordered arbitrarily. Fourth, category nodes and their membership edges are added. The result is a directed acyclic graph with 365 specialization and 281 prerequisite edges over 214 principles, plus 14 category nodes and 214 category edges. Appendix C details the procedure and two validations: a second model from a different family reproduces the edges with substantial agreement, and a manual audit finds 87% of edges correct.

We do not claim the graph is a complete or canonical account of how secure-coding principles relate. It is an operational structure: it exposes specific principles only after broader ones have been considered, and it avoids flat prompting over all 214 at once. Every candidate is still judged against the task and the current program, so the graph guides reasoning rather than acting as an oracle, and its measured benefit is to shorten that reasoning rather than to improve its outcome (Appendix F).

3.2 Structured SCP Reasoning

Given a task s, the SCP Graph G, a relevance threshold τ and a batch bound `max_size`, GRASP incrementally refines an initial solution. Let c_i denote the program after i rounds.

Step 1: Seed generation. GRASP drafts a seed solution c_0 from a prompt that asks for a short analysis of inputs, APIs and assumptions before the code, and instructs the model to add no security logic beyond what the task requires. The security content of the final program therefore comes from the rounds that follow rather than from the draft.

Step 2: Category selection. Using the task and the seed, GRASP scores each of the 14 categories on a 1–5 relevance scale and retains those scoring at least τ. Each category corresponds to a high-level concern such as input validation, error handling or file management, so this step narrows reasoning to the regions of G likely to contain applicable obligations. The scores are well spread rather than concentrated above the threshold, and at $\tau = 3$ the step prunes 60% of categories, selecting 5.55 of 14 per task on average (Appendix C).

Step 3: Dependency-aware selection. Within the selected categories, GRASP identifies the *ready* principles: those with no parent in the selected subgraph, or whose parents have all been visited with at least one of them applied. From this ready set it takes at most `max_size` principles for the current round, which bounds prompt length and stops a single round applying too many obligations.

Step 4: Application. For each principle in the batch, the model judges whether the obligation bears on the current code and task, and returns a decision with a reason rather than a numeric score. When it applies, the code is revised to satisfy the obligation, giving c_{i+1}; the revision prompt instructs the model to preserve the original functionality, so functional correctness is considered during each refinement rather than as a post-processing step. When it does not apply, the principle is marked visited and the code is unchanged, so the branch beneath it closes unless another parent is applied. This pruning is what keeps the walk from visiting all 214 principles, and recording each verdict with its reason leaves an inspectable trace of what was applied and what was dismissed.

Step 5: Expansion and termination. After the batch, GRASP adds newly ready children to the candidate set and repeats. Reasoning terminates when no ready principles remain, and the final program c_f is the result of incrementally applying the task-relevant obligations. Every round marks at least one principle visited, so the walk halts after at most $|V_0|$ rounds, where V_0 is the candidate set induced by the selected categories. In practice a sample costs 7.5 rounds on average. Figure 2 shows the traversal on a concrete task, a Flask route that loads a file named by a query parameter. Appendix D gives the pseudocode and Appendix E works it through in full.

4 Experiments

4.1 Setup

Benchmarks. Existing benchmarks are inadequate for evaluating secure code generation in realistic settings for two reasons. Security datasets (He & Vechev, 2023; He et al., 2024; Nazzal et al., 2024) focus on vulnerability-rich code but lack unit tests, so they cannot tell whether functionality survives; correctness datasets such as HumanEval (Chen et al., 2021) measure functional accuracy but omit security. Benchmarks that do evaluate both differ in prompt form or in oracle coverage: CWEval (Peng et al., 2025) focuses on code completion, where the model completes partially written

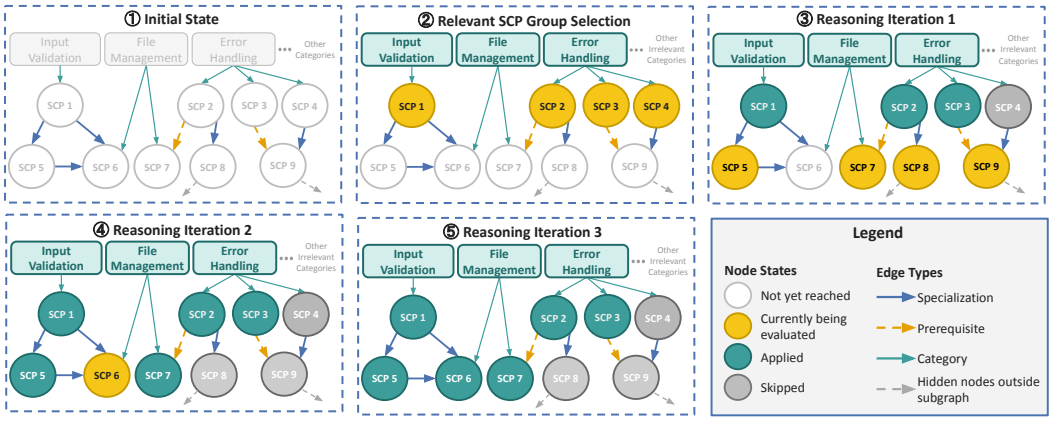

Figure 2: End-to-end SCP Graph traversal for the Flask file-loading route. ① shows the initial state, ② shows category selection, and ③–⑤ show reasoning rounds 1–3, with nodes colored by state.

code, and BaxBench (Vero et al., 2025) evaluates full backend application generation under specified framework and deployment constraints. Three further benchmarks share our intent but not our setting. SeCodePLT (Nie et al., 2025) targets the same joint evaluation at much larger scale, 1,411 Python tasks, but its C and C++ tasks carry no functional tests, and we wanted both oracles present in more than one language. RealSec-bench (Wang et al., 2026) and SecRepoBench (Shen et al., 2026) evaluate completion inside real repositories, 105 Java instances and 318 C and C++ tasks judged by a static pipeline and by the repositories' own fuzzing and unit tests respectively; both target repository-scale completion over long inter-procedural flows rather than standalone generation from a prompt. We therefore curate an evaluation set of natural-language prompts. A *scenario* defines the coding task and its *prompt* specifies how the task is presented. Unlike completion datasets, these prompts introduce additional challenges for correctness evaluation, including variation in names, arguments, return types and structure; to keep evaluation comparable, each prompt specifies function signatures, argument types, return values and I/O behavior. The set consists of 54 prompts, 37 Python and 17 C, adapted from prior work (Hajipour et al., 2024; Pearce et al., 2021; He & Vechev, 2023) and covering 17 CWEs. Python scenarios are evaluated with `pytest` (pytest development team, 2024) and C scenarios through compilation and shell scripts. Appendix B gives details and an example prompt. We also evaluate on CWEval and BaxBench to test generalization beyond our own data.

Models. We evaluate five LLMs spanning proprietary and open-weight families: OpenAI's gpt-4o-mini and gpt-5-nano (OpenAI, 2024c), Anthropic's Claude-4.5-Haiku (Anthropic, 2024a), and Alibaba's Qwen2.5-14B (Qwen et al., 2025) and Qwen3-Coder-30B (Yang et al., 2025). For readability we refer to them as GPT4o, GPT5, Claude4.5, Qwen2.5 and Qwen3. We use each provider's default decoding settings unless otherwise specified.

Metrics. We report three metrics. *Security Rate* (SR) is the proportion of generated samples that pass our security checks. A sample counts as secure only if it is syntactically valid and no applicable analyzer reports a vulnerability, which is stricter than a single-analyzer criterion:

$$\mathrm{SR} := \frac{\text{secure and valid samples}}{\text{total samples}}. \tag{1}$$

pass@k measures the probability that at least one of k samples is functionally correct. Letting c be the number of samples that pass all unit tests, and treating code that is syntactically invalid or fails to compile as functionally incorrect, we use the unbiased estimator of Chen et al. (2021):

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

Table 1: Security Rate, pass@k and secure-pass@k on our dataset, for the unmodified model (Base) and GRASP, across five LLMs at $n = 25$, split by language. **Bold** marks the better of the two for each metric. GRASP improves both the security rate and the joint security-and-correctness outcome at every k and for every model, in both languages. p@k abbreviates pass@k.

Model	Method	Python							C						
		SR	p@1	p@5	p@25	sp@1	sp@5	sp@25	**SR**	p@1	p@5	p@25	sp@1	sp@5	sp@25
GPT4o	Base	0.36	**0.83**	**0.89**	**0.89**	0.24	0.33	0.43	0.68	0.78	**0.87**	**0.88**	0.52	0.71	0.76
	GRASP	**0.79**	0.70	0.82	0.84	**0.55**	**0.71**	**0.78**	**0.90**	**0.81**	**0.87**	**0.88**	**0.72**	**0.87**	**0.88**
GPT5	Base	0.45	**0.80**	**0.88**	**0.92**	0.33	0.43	0.57	0.77	**0.86**	**0.88**	**0.88**	0.64	0.76	0.76
	GRASP	**0.79**	0.70	0.81	0.86	**0.55**	**0.71**	**0.81**	**0.92**	0.81	0.87	**0.88**	**0.74**	**0.85**	**0.88**
Claude4.5	Base	0.38	**0.78**	**0.87**	**0.89**	0.26	0.32	0.41	0.74	**0.81**	0.84	**0.88**	0.61	0.66	0.71
	GRASP	**0.70**	0.71	0.86	**0.89**	**0.46**	**0.64**	**0.73**	**0.84**	0.79	**0.86**	**0.88**	**0.69**	**0.81**	**0.82**
Qwen2.5	Base	0.43	**0.71**	**0.78**	**0.84**	0.24	0.33	0.46	0.77	**0.79**	0.87	**0.88**	0.62	0.75	0.76
	GRASP	**0.63**	0.46	0.73	**0.84**	**0.27**	**0.44**	**0.54**	**0.87**	0.75	**0.88**	**0.88**	**0.64**	**0.82**	**0.82**
Qwen3	Base	0.45	**0.83**	**0.86**	**0.86**	0.32	0.37	0.41	0.73	**0.87**	**0.88**	**0.88**	0.64	0.70	0.71
	GRASP	**0.66**	0.70	0.83	**0.86**	**0.44**	**0.56**	**0.70**	**0.84**	0.82	**0.88**	**0.88**	**0.69**	**0.79**	**0.88**

secure-pass@k applies the same estimator to the stricter event, requiring a sample to be both functionally correct and secure. Letting sp be the number of samples that pass all unit tests and contain no detected vulnerability, we follow Fu et al. (2024):

$$\text{secure-pass@}k := \mathbb{E}_{\text{scenarios}}\left[1 - \frac{\binom{n-sp}{k}}{\binom{n}{k}}\right]. \tag{3}$$

We abbreviate pass@k as p@k and secure-pass@k as sp@k. SR alone is not sufficient, because a program that does nothing passes most security checks; we therefore use sp@k as the primary metric, and SR is reported alongside it for continuity with prior work. Following prior work (Pearce et al., 2021; He & Vechev, 2023) we generate $n = 25$ samples per scenario unless stated otherwise, and detect vulnerabilities with CodeQL (CodeQL, 2024) using targeted queries for each scenario's CWE, extended with Bandit (OpenStack Security Group, 2024) for Python and Cppcheck (Klamer & the Cppcheck developers, 2025) for C.

Baselines. We compare against **ZS**, zero-shot security prompting that prepends a security-focused instruction; **PaS**, planning-based prompting that asks the model to plan before generating; **RAG**, retrieval-augmented prompting over security documentation; **PromSec** (Nazzal et al., 2024), analyzer-guided prompt optimization; and **RESCUE** (Shi & Zhang, 2026), retrieval of distilled security guidelines and sliced secure code examples from a knowledge base built offline from vulnerability-fix pairs. The set spans the families these defenses fall into: prompting with ZS and PaS, retrieval at inference time with RAG and RESCUE, and iterative refinement under feedback with PromSec, so the comparison is not confined to one kind of defense. Isolating the contribution of GRASP's own components is the job of the ablations in Appendix F.

GRASP configuration. We use $\tau = 3$ and `max_size` $= 8$ uniformly across models as model-agnostic defaults rather than per-model optima: the sweeps in Appendix F show GPT5 scoring higher at $\tau = 2$ and at `max_size` $= 16$, so these are not the best setting for every model.

4.2 SECURITY AND FUNCTIONAL CORRECTNESS

GRASP raises security across every model. Table 1 shows consistent SR improvements. The gains are largest on Python, where the base models are weakest: GPT4o improves from 0.36 to 0.79, GPT5 from 0.45 to 0.79, and Claude4.5 from 0.38 to 0.70, with the open-weight models rising from 0.43 to 0.63 and from 0.45 to 0.66. On C the base rates are already between 0.68 and 0.77, and GRASP raises them to between 0.84 and 0.92. Gains are therefore largest where the base model is weakest, but even the strongest model we evaluate still emits vulnerable Python roughly 55% of the time without intervention, so the need does not disappear with model capability.

Table 2: GPT5 comparison across our dataset, CWEval and BaxBench, reporting p@k and sp@k. Restricted to the Python subset of each benchmark, since PromSec is Python-only. **Bold** marks the best value in each column, underline the second best.

Method	Our Dataset				CWEval				BaxBench			
	p@1	p@5	sp@1	sp@5	p@1	p@5	sp@1	sp@5	p@1	p@5	sp@1	sp@5
ZS	0.65	0.86	0.37	0.68	0.90	**1.00**	0.87	**1.00**	0.51	0.74	**0.40**	0.61
PaS	**0.74**	0.89	0.49	0.70	0.94	**1.00**	**0.94**	**1.00**	0.48	0.76	0.36	0.59
RAG	0.71	0.89	0.43	0.65	0.94	**1.00**	**0.94**	**1.00**	0.25	0.40	0.21	0.36
PromSec	0.14	0.27	0.09	0.19	0.51	0.72	0.46	0.68	0.03	0.13	0.01	0.08
RESCUE	0.64	**0.92**	0.35	0.59	0.90	**1.00**	0.86	**1.00**	**0.55**	0.78	0.38	0.61
GRASP	0.71	0.81	**0.56**	**0.76**	**0.97**	**1.00**	0.93	**1.00**	0.46	**0.81**	0.37	**0.73**

The improvements are not confined to one vulnerability family. GRASP is especially effective on CWE-078, where every model reaches an SR of at least 0.88 and four reach 0.94 or higher, which suggests principle-guided reasoning helps most where secure generation requires recognizing a dangerous data flow and enforcing constraints before a sensitive call. It also substantially improves CWE-022, where path traversal defenses require reasoning about filesystem boundaries, path normalization and untrusted input. On the remaining 38 scenarios covering 14 other CWEs, GRASP improves SR across all models, reaching 0.69–0.81. Appendix G gives the full per-CWE breakdown.

The security gain costs single-sample correctness, not reach. Functional correctness at pass@1 falls for every model on Python, from 0.83 to 0.70 for GPT4o and most sharply from 0.71 to 0.46 for Qwen2.5, so an individual GRASP sample is more likely to be functionally wrong than a Base sample. That cost does not persist as samples accumulate: by pass@25 the two are within 0.06 on Python and identical on C, so GRASP lowers the per-sample hit rate rather than putting tasks beyond reach.

The joint outcome improves regardless. Because SR credits a sample that the analyzers accept whether or not it runs, the security result above is only meaningful alongside sp@k, which requires both. Across all five models, GRASP improves sp@k over Base at every evaluated k (Table 1). The improvement is already visible at $k = 1$, which matters because secure generation is most useful when the first returned solution is both correct and safe, and it persists as k grows: because the security gain is far larger than the functional cost, the two net out in GRASP's favor at every k and every model family.

4.3 COMPARISON WITH PROMPTING AND RETRIEVAL BASELINES

Table 2 compares GRASP with prompting, retrieval-based and analyzer-guided baselines on GPT5, at $n = 5$ samples per scenario.

Structured principle application improves joint security and correctness on our dataset. GRASP achieves the highest sp@1 and sp@5, reaching 0.56 and 0.76, compared with 0.49 and 0.70 for PaS, the strongest baseline. ZS, RAG and RESCUE cluster between 0.35 and 0.43 at sp@1, so a security instruction or retrieved guidance carries a method part of the way but not the whole distance. RESCUE shows the limit of supplying that information without applying it in context: it reaches the highest pass@5 of any method, 0.92 against GRASP's 0.81, so it leaves the program largely intact, yet its sp@1 of 0.35 places it with the single-pass prompting baselines. It regenerates once from knowledge organized around weakness classes, so a weakness its corpus does not name goes undefended. The additional gains from GRASP therefore suggest that how guidance is applied matters alongside its availability, and the ablations in Appendix F provide more direct evidence: replacing iterative refinement with a single prompt containing selected principles reduces joint security and correctness. PromSec sits far below every other method, and its shortfall is structural rather than a security failure: functional correctness collapses to 0.14 at p@1, and across its runs code that will not import or parse outnumbers failed assertions by about four to one. Optimizing the prompt against a security objective also rewrites the contract the task stated, so the program drifts from the interface the unit tests expect, though another optimizer configuration may fare better.

GRASP broadens task coverage on BaxBench under multiple sampling. On BaxBench, GRASP sits mid-field at sp@1, at 0.37 against 0.40 for ZS and 0.38 for RESCUE, but leads clearly at sp@5, reaching 0.73 where both of those reach 0.61. With $n = 5$, sp@5 is the fraction of tasks where at least one candidate passes both checks, so the benefit appears when several candidates can be evaluated rather than when only one is generated.

Near-saturated performance limits gains on CWEval. On CWEval's Python subset every method except PromSec reaches 1.00 at sp@5 and sp@1 spans only 0.86 to 0.94, so there is little headroom for additional guidance: GRASP's 0.93 at sp@1 sits marginally behind PaS and RAG at 0.94. This contrasts with our dataset and BaxBench: additional refinement is not uniformly beneficial when the initial success rate is high.

4.4 GENERALIZATION TO UNSEEN VULNERABILITIES

Table 3: Aggregate Security Rate on the six CVEs disclosed after both models' training cutoffs.

	GPT4o				GPT5			
Method	Total	Valid	Secure	SR	Total	Valid	Secure	SR
ZS	150	150	12	0.08	150	150	3	0.02
PromSec	150	150	6	0.04	150	105	1	0.01
RESCUE	150	150	37	0.25	150	150	29	0.19
GRASP	150	150	69	**0.46**	150	150	75	**0.50**

The results so far are measured on benchmarks whose weaknesses the evaluated models may have seen during training. We therefore turn to vulnerabilities disclosed after a model's training cutoff, which we call zero-day for that model. A disclosure date after the cutoff does not prove the surrounding code or discussion was absent from training, so this reduces contamination risk rather than eliminating it. We evaluate on Python CVEs collected from GitHub's CodeQL Wall of Fame. These were originally reported as CodeQL-discovered issues, so we use CodeQL as the security oracle to determine whether generated code still exhibits the corresponding vulnerability, which gives a controlled way to test whether each method mitigates the same pattern CodeQL identified in the original project. Each model is evaluated only on CVEs disclosed after its own training cutoff. The CVE prompts carry no unit tests, so sp@k is undefined here and SR is the only metric the setting supports.

GRASP mitigates zero-day weaknesses far more often than any baseline. On the six CVEs disclosed after both models' cutoffs, GRASP reaches an SR of 0.46 on GPT4o and 0.50 on GPT5, against 0.25 and 0.19 for RESCUE, the best baseline, and 0.08 and 0.02 for zero-shot security prompting (Table 3). The margin is widest where the weakness maps onto a transferable obligation such as validating untrusted input or constraining filesystem access: on CVE-2024-39685 and CVE-2024-39686 GRASP reaches 0.96 for GPT4o, and 0.56 and 0.68 for GPT5, where no baseline exceeds 0.28 on either model. A corpus assembled before disclosure cannot name these flaws, so what carries over is the obligation rather than any description of the weakness, which is what a principle-centric defense predicts and a vulnerability-centric one does not.

The aggregate gain is not a uniform one. Across the 20 CVE prompts GRASP leads or ties on 11. Three of the nine it does not lead are already solved outright by zero-shot prompting, leaving nothing to add; on four of the rest the winner is RESCUE, whose corpus evidently covers the weakness class, as on CVE-2025-49833 for GPT4o where it reaches 1.00 against 0.76, although GRASP reaches 1.00 on that same CVE for GPT5. The 2025 CVEs are hard for everything we evaluate. Both observations fit the same reading: an obligation transfers when the weakness can be stated as one, and less well when it turns on the behavior of a particular library, where vulnerability-specific knowledge still wins. Appendix H gives the per-CVE breakdown and the remaining CVEs.

4.5 ADDITIONAL ANALYSIS

Three further questions bear on whether structured reasoning earns its cost, and we take each up in the appendix. First, is an inference-time method redundant once a model has been trained for

security? Appendix I compares two security fine-tuned checkpoints on the same base and finds fine-tuning alone not consistently superior, and combining it with GRASP mixed rather than additive, so the two are not substitutes. Second, which components carry the result? Appendix F removes each in turn and sweeps both hyperparameters, locating the quality gain in how principles are applied rather than merely supplied, and the graph's contribution in efficiency. Third, what does the reasoning cost? Appendix J measures tokens and steps against PromSec and finds the trade-off model dependent: on GPT5 GRASP spends moderately more for substantially higher sp@1, while on Qwen3 it is both cheaper and more effective.

5 DISCUSSION

Dependence on model capability. GRASP operates through inference-time structured reasoning rather than fine-tuning, which makes it model-agnostic but leaves its effectiveness dependent on the model's ability to follow multi-step instructions, evaluate principle relevance, and revise code while preserving functionality. Our results reflect this: proprietary models achieve the highest absolute performance while open-weight models improve from a lower base. The gains are not confined to the strongest models, so the approach should become more practical as open-weight models improve.

Extending and updating principle sources. We instantiate the graph from the OWASP checklist because it provides broad, language-agnostic guidance, but the construction pipeline is source-agnostic and can absorb language-specific standards such as CERT C or CERT Java (CERT, n.d.). Because principles are used at inference time, new ones take effect without retraining.

Principles transfer past their own coverage. The 214 principles state obligations rather than name weaknesses, so they do not line up one-to-one with the CWEs we measure. Fourteen of the 17 in our set have a corresponding obligation and three, CWE-416, CWE-611 and CWE-918, have none. Those three still improve. XXE is the clearest case, rising from SR 0.02 to 0.66 with no principle anywhere in the graph that mentions XML parsing, apparently through the general obligations about validating untrusted input and constraining access to external resources. We read this as the same mechanism behind the post-cutoff CVE results: what transfers is the obligation, not a description of the flaw, which is also why we would not expect the method to help where a weakness turns on the behavior of one particular library.

Cost of the reasoning. GRASP replaces one generation call with 7.5 on average, a seed plus one per batch of principles. Against PromSec, the other iterative method, it spends 10% more input and 20% more output tokens on GPT5 and about a third as many on Qwen3 (Appendix J). The cost is reasoning calls rather than training runs or analyzer infrastructure, which makes the method better suited to generation than to latency-sensitive completion.

Reliance on static analysis. Our primary security metric is analyzer-based, which provides a scalable and reproducible oracle but should not be read as proof that generated code is vulnerability-free, since static analyzers may miss vulnerabilities requiring deeper semantic or runtime reasoning. We partially mitigate this by using multiple analyzers rather than one, making the criterion more conservative than any individual tool, and we evaluate on external benchmarks whose oracles are independent of our analyzer suite.

6 CONCLUSION

We presented GRASP, which shifts secure code generation from reactive patching toward prescriptive prevention by iteratively applying the secure-coding principles a task raises, organized as a dependency graph that keeps the reasoning short. It raises security across five models, improves secure generation on every benchmark that leaves headroom, and transfers to zero-day CVEs, at a functional cost that shows on a single sample and closes as samples accumulate. Extending it to coding agents (Yang et al., 2024; Anthropic, 2025; OpenAI, 2025), where long horizons compound insecure edits (Sajadi et al., 2025) and admit prompt injection (Przymus et al., 2025), is a natural next step.

Ethics Statement

This work aims to reduce the incidence of vulnerabilities in machine-generated code, and we believe its disclosure risk is low. It introduces no new attack technique and reports no previously unknown vulnerability. The CVEs used in Section 4.4 are public, already patched by their upstream projects, and drawn from a published collection; our prompts restate the function each patch touched rather than supplying a working exploit. The benchmarks we evaluate on are released by their authors for this purpose and we use their harnesses unmodified.

Two considerations deserve stating plainly. A method that improves the security of generated code can induce misplaced confidence: none of our results should be read as a guarantee that code produced with GRASP is free of vulnerabilities, and Section 5 bounds what our oracles can establish. A defense that is right most of the time still requires review before code reaches production. Second, principle-guided reasoning is symmetric in principle, since the same structure that directs a model toward secure implementations could direct it elsewhere; our contribution supplies no capability a model lacks without it, and organizes publicly documented guidance. No human subjects were involved and no personal data was collected.

Reproducibility Statement

Code, the SCP Graph, every prompt and the scripts that produce each table are available at `https://anonymous.4open.science/r/GRASP_ICLR-09C7`. The paper is written so the experiments can be repeated without the repository as well. Section 3 specifies the method including $\tau = 3$ and `max_size` $= 8$, and Appendix D gives the procedure as pseudocode; every prompt is released verbatim in the repository. Section 4.1 defines all three metrics as equations and records the benchmarks, model versions, decoding settings, sampling budget and baseline configurations. Appendix C describes how the graph was constructed. Two sources of variation are outside our control: the proprietary models are served behind APIs whose weights may change without notice, and generation is stochastic, so individual samples differ between runs while the reported metrics are averages over n samples per scenario.

AI Use Statement

Large Language Models are the object of study in this work and their use as such is described throughout. Separately, the authors used LLMs to aid and polish prose for clarity. The construction of the SCP Graph is itself LLM-assisted by design, as Section 3 and Appendix C describe; that is a component of the method rather than an authoring aid. All research questions, experimental design decisions, analyses and claims are the authors' own, and the authors have verified every result reported and take full responsibility for the contents.

References

Code generation on humaneval leaderboard. `https://paperswithcode.com/sota/code-generation-on-humaneval`, 2024. Accessed: 2024-09-01.

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)*, pp. 618–625, 2018. doi: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00112.

Anthropic. Claude models documentation, 2024a. URL `https://docs.anthropic.com/en/docs/about-claude/models`. Accessed: 2024-09-03.

Anthropic. Claude. `https://claude.ai/`, 2024b. Accessed: 2024-07-29.

Anthropic. Claude code. `https://www.anthropic.com/claude-code`, 2025. Accessed: 2026-05-03.

Owura Asare, Meiyappan Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code?, 2024. URL `https://arxiv.org/abs/2204.04741`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

CERT. Cert secure coding standards, n.d. URL https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards. Accessed: 2024-08-04.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, 15(3):1–45, 2024.

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. URL https://arxiv.org/abs/2107.03374.

CodeQL. Codeql, 2024. URL https://codeql.github.com/. Accessed: 2024-08-31.

DeepMind. Gemini, 2024. URL https://deepmind.google/technologies/gemini/#introduction. Accessed: 2024-08-31.

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. URL https://arxiv.org/abs/2002.08155.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. Constrained decoding for secure code generation, 2024. URL https://arxiv.org/abs/2405.00218.

GitHub. Github copilot, 2024. URL https://github.com/features/copilot. Accessed: 2024-08-04.

Google Research. Ml-enhanced code completion improves developer productivity, 2024. URL https://research.google/blog/ml-enhanced-code-completion-improves-developer-productivity/. Accessed: 2024-08-31.

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)*, pp. 684–709. IEEE, 2024.

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. doi: 10.1145/3576915.3623175. URL http://dx.doi.org/10.1145/3576915.3623175.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation, 2024. URL https://arxiv.org/abs/2402.09497.

IBM. Ibm data breach reports, 2024a. URL https://www.ibm.com/reports/data-breach. Accessed: 2024-08-31.

IBM. Reputational it risk report, 2024b. URL https://images.forbes.com/forbesinsights/StudyPDFs/IBM_Reputational_IT_Risk_REPORT.pdf?ref=blog.cubbit.io. Accessed: 2024-08-31.

Eirini Kalliamvakou. Research: Quantifying github copilot's impact on developer productivity and happiness, 2024. URL https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/. Accessed: 2024-08-04.

Julien Klamer and the Cppcheck developers. Cppcheck: A tool for static c/c++ code analysis, 2025. URL https://cppcheck.sourceforge.io/. Accessed: 2025-11-12.

Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. INDICT: Code generation with internal dialogues of critiques for both security and helpfulness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/9b812ee4b831c21e14156ced8659197c-Abstract-Conference.html.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

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*, pp. 152–156, 2012. doi: 10.1109/MINES.2012.202.

Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet A. Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, Yuxiang Wei, Lingming Zhang, Ismini Lourentzou, and Gang Wang. Purpcode: Reasoning for safer code generation, 2025. URL https://arxiv.org/abs/2507.19060.

Na Meng, Stefan Nagy, Daphne Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: Challenges and vulnerabilities, 2017. URL https://arxiv.org/abs/1709.09970.

Meta. Llama, 2024. URL https://llama.meta.com/. Accessed: 2024-08-31.

Microsoft. Secure coding guidelines, 2022. URL https://learn.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines. Accessed: 2024-08-04.

MITRE. Cwe top 25 most dangerous software errors: 2023 edition, 2023. URL https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. Accessed: 2024-08-31.

MITRE. Common weakness enumeration (cwe), 2024. URL https://cwe.mitre.org/index.html. Accessed: 2024-08-31.

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. URL https://arxiv.org/abs/2505.10494.

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*, pp. 2266–2280, 2024.

Yuzhou Nie, Zhun Wang, Yu Yang, Ruizhe Jiang, Yuheng Tang, Xander Davies, Yarin Gal, Bo Li, Wenbo Guo, and Dawn Song. SeCodePLT: A unified platform for evaluating the security of code GenAI. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2410.11096.

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. URL https://arxiv.org/abs/2203.13474.

OpenAI. Openai models documentation. https://platform.openai.com/docs/models, 2024a. Accessed: 2024-07-29.

OpenAI. Openai codex, 2024b. URL https://openai.com/index/openai-codex/. Accessed: 2024-08-31.

OpenAI. Openai api models documentation, 2024c. URL https://platform.openai.com/docs/models. Accessed: 2024-09-03.

OpenAI. OpenAI codex. https://openai.com/codex/, 2025. Accessed: 2026-05-03.

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

OWASP Foundation. Owasp secure coding practices, n.d. URL https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/02-checklist/05-checklist. Accessed: 2024-08-04.

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. URL https://arxiv.org/abs/2108.09293.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation, 2025. URL https://arxiv.org/abs/2501.08200.

Piotr Przymus et al. Adversarial bug reports as a security risk in language model-based automated program repair. 2025. arXiv:2509.05372.

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

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

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. URL https://arxiv.org/abs/2403.12671.

Amirali Sajadi et al. How safe are AI-generated patches? a large-scale study on security risks in LLM and agentic automated program repair on SWE-bench. 2025. arXiv:2507.02976.

Chihao Shen, Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. SecRepoBench: Benchmarking code agents for secure code completion in real-world repositories. In *Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code)*, 2026. doi: 10.1145/3786181.3788703. URL https://dl.acm.org/doi/full/10.1145/3786181.3788703.

Jiahao Shi and Tianyi Zhang. RESCUE: Retrieval augmented secure code generation. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=gbxhesw4UH.

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

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. Baxbench: Can llms generate correct and secure backends?, 2025. URL https://arxiv.org/abs/2502.11844.

Yanlin Wang, Ziyao Zhang, Chong Wang, Xinyi Xu, Mingwei Liu, Yong Wang, Jiachi Chen, and Zibin Zheng. RealSec-bench: A benchmark for evaluating secure code generation in real-world repositories. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026. URL https://aclanthology.org/2026.findings-acl.1786/.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. Prosec: Fortifying code llms with proactive security alignment, 2025. URL https://arxiv.org/abs/2411.12882.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.15793.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. SecCoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 14557–14571, 2024. URL https://aclanthology.org/2024.emnlp-main.806/.

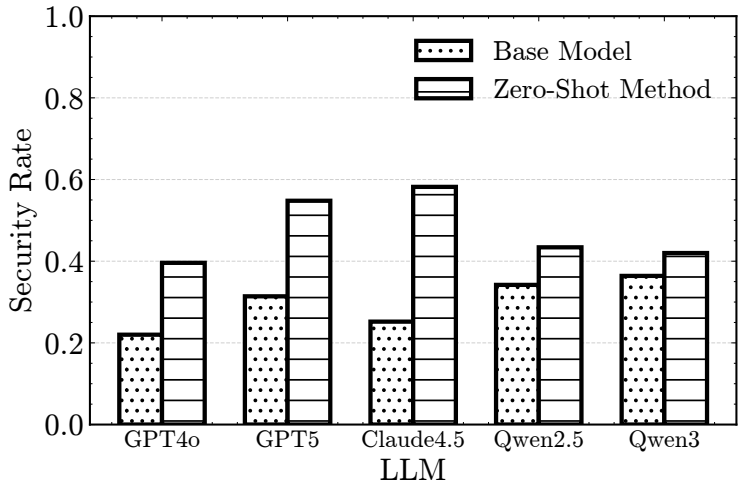

Figure 3: SR of Base Model and Zero-Shot Method.

A Motivation and Observation

A.1 Risk of LLM-Based Code Generation

To understand the risk of LLMs in code generation, we conducted a preliminary quantitative study to examine their tendency to produce vulnerable code. For this updated study, we selected a hard 20-scenario Python subset from our dataset whose scenarios are challenging for both Base prompting and Zero-Shot prompting. The selected subset spans 8 CWEs, including path traversal, command injection, insecure deserialization, open redirect, XML, XSS, information exposure, XPath injection, permission issues, and SSRF.

We evaluated two prompting methods: (1) directly querying the Base Model with the original prompts, and (2) employing a Zero-Shot prompt that incorporates an additional security-focused instruction: *"You are a Security Engineer, and you develop code that adheres to secure coding principles."*. We evaluated OpenAI's gpt-4o-mini (GPT4o) and gpt-5-nano (GPT5) (OpenAI, 2024c), Anthropic's Claude-4.5-Haiku (Claude4.5) (Anthropic, 2024a), and the open-weight models Qwen2.5-14B (Qwen2.5) and Qwen3-Coder-30B (Qwen3). Following prior work (He & Vechev, 2023; Pearce et al., 2021), we generated 25 samples with each prompt across all LLMs and prompt types.

To assess the presence of vulnerabilities in the generated samples, we employed two widely used code analysis tools: CodeQL (CodeQL, 2024) and Bandit (OpenStack Security Group, 2024). A sample was considered vulnerable if any of these tools flagged a vulnerability. The evaluation metric adopted is Security Rate (SR), defined as the proportion of samples that are secure, as detailed in Equation 1 in Section 4.1. Notably, we do not explore false negatives, as our focus is on identifying potential risks in the generated code rather than ensuring complete vulnerability detection coverage.

As shown in Figure 3, all five models consistently generate vulnerable code on this harder subset. The Base models achieve SRs ranging from 0.22 to 0.36, with GPT4o performing worst at 0.22 and Qwen3 performing best at 0.36. Zero-Shot security prompting improves SR across all evaluated models, with the strongest result reaching 0.58 for Claude4.5; however, these gains remain limited and still leave a substantial fraction of generated code vulnerable. These findings suggest that simply instructing models to be security-aware is insufficient, motivating our approach of guiding models through structured reasoning over SCPs to identify and apply the security principles that govern a programming task.

Independent evaluations at repository scale reach the same conclusion from a different direction. SecRepoBench (Shen et al., 2026) poses 318 completion tasks across 27 C and C++ repositories, judged by the repositories' own fuzzing and unit tests, and RealSec-bench (Wang et al., 2026) covers 105 Java tasks whose vulnerabilities span inter-procedural flows of up to 34 hops; both find that models continue to emit vulnerable code once the surrounding project is in scope, so the problem is

not an artifact of short standalone prompts. RealSec-bench additionally reports two results that bear on the remedies. Retrieval augmentation improves functional correctness while adding little security, and prompting with general security guidelines frequently causes compilation failures, degrading correctness without reliably preventing the vulnerabilities those guidelines describe. Undirected guidance is therefore not a free intervention, which sharpens the gap we target: the question is not how much security knowledge reaches the model, but which obligations reach it and in what order.

A.2 Challenges with Existing Approaches

While existing techniques for fortifying LLM-based code generation (He & Vechev, 2023; He et al., 2024; Liu et al., 2025; Xu et al., 2025; Nazzal et al., 2024) have demonstrated measurable security improvements, they face four fundamental limitations that hinder broad applicability and long-term maintainability.

Dependence on Large Curated Datasets: SVEN (He & Vechev, 2023) and SafeCoder (He et al., 2024) require large, curated datasets of vulnerability-fix pairs from historical code. ProSec (Xu et al., 2025) synthesizes vulnerability scenarios from CWEs to create datasets significantly larger than prior work, while PurpCode (Liu et al., 2025) requires comprehensive cybersafety datasets for its two-stage alignment. PromSec (Nazzal et al., 2024), though not tuning the LLMs, relies on vulnerability data to train a gGAN optimizer. This vulnerability-centric paradigm, which teaches models what insecure patterns to avoid, requires continuous dataset curation and retraining as new vulnerability patterns emerge, limiting scalability and maintainability. PurpCode partially addresses this by teaching explicit cybersafety rules in its first stage; however, these rules are fundamentally derived from CWEs and code analyzer vulnerability patterns (e.g., CodeGuru, CodeQL) rather than prescriptive secure coding principles. While expressed as rules to follow, they remain vulnerability-centric in nature, focusing on what patterns to avoid. Moreover, PurpCode still relies on vulnerability-inducing scenarios for training and reinforcement learning in its second stage, maintaining its dependence on vulnerability data. Retrieval-based methods leave the weights alone but inherit the same dependence at the corpus level. RESCUE (Shi & Zhang, 2026) distills its knowledge base from vulnerability-fix pairs and SecCoder (Zhang et al., 2024) retrieves from a pool of safe demonstrations, so each still requires a curated collection to retrieve from and must have it re-curated as new patterns appear.

Poor Generalization to Unseen Vulnerabilities: Although effective on previously seen patterns, vulnerability-centric approaches are tightly coupled to their training data. Zero-day vulnerabilities pose a severe challenge, as these systems cannot detect or mitigate patterns they have never encountered. The evolving nature of security threats demands methods that can reason from secure-coding principles rather than memorized vulnerability patterns, enabling adaptation to novel attacks without requiring retraining on new exploit data. Retrieval shifts this limitation rather than removing it: what RESCUE and SecCoder can advise on is bounded by what their corpus contains, so a weakness absent from it is never surfaced at generation time.

Impracticality for Proprietary Models: Tuning-based methods (SVEN (He & Vechev, 2023), SafeCoder (He et al., 2024), PurpCode (Liu et al., 2025), ProSec (Xu et al., 2025)) require access to model weights for prefix-tuning, instruction tuning, or reinforcement learning, making them incompatible with closed-source LLMs such as GPT4o or Claude4.5. PromSec (Nazzal et al., 2024), while compatible with black-box models, requires training a separate gGAN model and depends on external static analyzers (Bandit (OpenStack Security Group, 2024), SpotBugs (SpotBugs Contributors, 2025)) for iterative feedback during inference.

These dependencies introduce infrastructure overhead, runtime costs, and performance limitations tied to external tool coverage. Additionally, the tuning-based nature of these approaches severely limits deployment in real-world settings where only API access is available. Retrieval-based and critique-based methods are not subject to this limitation, since RESCUE, SecCoder and INDICT (Le et al., 2024) all operate through prompting alone. INDICT, however, draws on execution and tool feedback in its post-hoc stage, which reintroduces the infrastructure requirement that analyzer-guided methods carry.

Limited Interpretability: Existing approaches provide limited visibility into how or why specific security transformations are applied. Tuning-based methods learn security patterns through gradient-based updates, making it difficult to understand emergent behaviors. PromSec's gGAN-based optimization operates through opaque adversarial training. PurpCode's Reinforcement Learn-

Table 4: Comparison of Existing Approaches.

Property	SVEN	SafeCoder	PurpCode	ProSec	PromSec	RESCUE	GRASP
Closed-source LLM compatible	✗	✗	✗	✗	✓	✓	✓
No training needed	✗	✗	✗	✗	∼	✓	✓
No vuln dataset required	✗	✗	✗	✗	✗	✗	✓
No external feedback tools	✓	✓	∼	✓	✗	✓	✓
Interpretable security reasoning	✗	✗	✗	✗	✗	∼	✓
Principle-centric approach	✗	✗	∼	✗	✗	∼	✓

Note: ✓ indicates that the model has the property; ∼ indicates partial or conditional support; ✗ indicates that it does not.

ing stage optimizes through reward signals that obscure the reasoning process, making it difficult to trace why specific security decisions are made during generation. This lack of full transparency poses challenges for developers, auditors, and security engineers who need to verify correctness and understand the reasoning behind code modifications. Retrieval-based methods are more legible in that the retrieved guidelines or demonstrations can simply be read, and INDICT's critiques are natural language, but none of them records a per-obligation decision tying a specific change to the specific requirement that motivated it, so it remains hard to see why a given transformation was made.

These challenges underscore the necessity of a new approach. Instead of teaching models what vulnerable patterns to avoid, we introduce GRASP, which guides models through structured reasoning over SCPs that specify what secure code should do and how it should satisfy task-relevant security obligations. GRASP addresses all four limitations: it requires no curated vulnerability datasets, generalizes to unseen vulnerabilities through principle-driven reasoning, works with any LLM, including closed-source models, without training or the use of external tools during inference, and provides interpretable security reasoning. Table 4 systematically compares GRASP against existing methods across these dimensions.

B DATASET DETAILS

The sample prompt in Figure 4 is structured according to the requirements outlined in Section 4.1. 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.

Table 5 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 5: 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

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"`

Figure 4: Sample scenario prompt from our dataset: a Flask route serving image files from a designated directory, with the endpoint, parameter and expected behavior for each branch stated explicitly.

Listing 1: Unit Test Case Example.

```
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
```

C SCP Graph Construction and Validation

Representation. Each principle is normalized into a structured record holding the principle text, its source category and metadata used during construction. Categories give coarse organization; the principles themselves are the reasoning units. We keep to obligations that can directly guide generation and exclude organizational or deployment guidance, which matters for security but cannot be enforced by editing a program.

Source and scale. We extract 214 principles from the OWASP Secure Coding Practices checklist (OWASP Foundation, n.d.), preserving their association with its 14 categories: Access Control, Authentication and Password Management, Communication Security, Cryptographic Practices, Data Protection, Database Security, Error Handling and Logging, File Management, General Cod-

ing Practices, Input Validation, Memory Management, Output Encoding, Session Management, and System Configuration.

Construction. The pipeline runs in four stages.

Stage 1, normalization. Each extracted principle is enriched into a structured description that clarifies the obligation and adds representative insecure and secure usage patterns together with relevant weakness categories. This stage is grounded in the extracted guidance, so the model enriches existing principles rather than inventing new ones, and it runs three LLMs independently over every principle, with the first combining their outputs.

Stage 2, pairwise edge detection. A single model judges each of the 22,791 unordered principle pairs once, classifying the relation as specialization, prerequisite, or none. This yields 888 edges, 443 specialization and 443 prerequisite, with two mutual cases and no parse failures.

Stage 3, structural resolution. The raw result is heavily cyclic, with 131 strongly connected components of which the largest spans 83 nodes. A model from a different family reviews the 406 pairs that are structurally suspect, meaning those participating in cycles or reciprocal edges, and resolves each. This prunes 27% of the inferred edges, leaving 648: 365 specialization, 281 prerequisite, and one pair the resolver declined to order, carried as two mutual-prerequisite edges. That pair is the only strongly connected component that survives, which is why the largest falls from 83 nodes to 2.

Stage 4, category attachment. The mutual pair is dropped rather than ordered arbitrarily, since either direction would assert a prerequisite the resolver had explicitly refused to commit to. This is where acyclicity is enforced, and it leaves 646 relation edges. The 14 category nodes and their 214 membership edges are then added, giving the shipped graph: 214 principles and 14 category nodes, 365 specialization edges, 281 prerequisite edges and 214 category edges, 860 in all, with no remaining cycles.

Category relevance scores. The threshold applies to a 1–5 score over the 14 categories rather than to individual principles. Across 11,024 (task, category) scores drawn from 795 traces over all five models, the distribution is well spread: 36.8% score 1, 23.2% score 2, 20.9% score 3, 12.7% score 4 and 6.4% score 5. At $\tau = 3$ the step prunes 60.0% of category groups and selects 5.55 of 14 categories per task on average, so it discriminates rather than admitting everything. Stronger models are more selective, with 29.2% of categories passing the threshold for Claude4.5 against 48.6% for Qwen3.

Validation. Two checks bear on whether the graph is an artifact of the model that built it. First, re-running the identical edge-detection prompt with a model from a different family over a stratified sample of 150 edge and 150 non-edge pairs, excluding the pairs the resolver had reviewed, gives raw agreement of 0.887 and a Cohen's κ of 0.773, or 0.759 when the edge type and direction must also match. The second model confirms 128 of 150 asserted edges and 138 of 150 non-edges, and where both call an edge they agree on type and direction in 109 of 128 cases. Because the sample is stratified against a population edge rate of 3.9%, this κ describes the stratified sample rather than the population. Second, a manual audit of 100 asserted edges found 87 correct, and of 100 unconnected pairs found 95 correctly unconnected. This audit was a single-rater pass with ambiguous items adjudicated by an author, so it reports one judgment rather than an inter-rater agreement, and we report it as a sanity check on edge quality rather than as independent validation.

We do not claim the graph is a complete or canonical account of how principles relate. It is an operational structure, and every candidate is still judged against the task and the current program before anything is applied.

D Structured Reasoning Algorithm

Algorithm 1 provides the full procedure for GRASP's structured SCP reasoning. Given a scenario s, the SCP Graph G_{SCP}, categories C, a relevance threshold τ, and a maximum batch size B, GRASP first drafts seed code c_0 from the seed prompt of Step 1. It then selects the categories most relevant to the task and restricts reasoning to the principles they contain.

The algorithm maintains two sets during reasoning: V_{visited}, which records SCPs that have already been evaluated, and V_{applied}, which records SCPs that were judged relevant and used to refine the

code. At each iteration, GRASP identifies the currently ready SCPs, packs up to B of them into a batch, and evaluates whether they apply to the current code. Relevant SCPs are applied through code refinement, while irrelevant or already-satisfied SCPs are marked as visited. The ready set is then recomputed from the graph, allowing GRASP to expand to newly available SCPs while pruning branches whose prerequisites were not applied. The process terminates when no ready SCPs remain, returning the final hardened code c_f.

Algorithm 1 Structured SCP Reasoning

Require: scenario s, SCP Graph G_{SCP}, categories C, relevance threshold τ, max batch size B
Ensure: security-hardened code c_f

$c_0 \leftarrow \textsc{GenerateSeedCode}(s)$ ▷ Step 1: Seed Code Generation
$C_{\text{sel}} \leftarrow \{c \in C \mid \textsc{ScoreGroup}(c, s, c_0) \geq \tau\}$
$S_{\text{sel}} \leftarrow \bigcup_{c \in C_{\text{sel}}} \textsc{Members}(c)$ ▷ Step 2: Relevant SCP Group Selection
$c_i \leftarrow c_0$
$V_{\text{visited}} \leftarrow \emptyset$; $V_{\text{applied}} \leftarrow \emptyset$
$Q \leftarrow \textsc{ReadyNodes}(G_{\text{SCP}}, S_{\text{sel}}, V_{\text{visited}}, V_{\text{applied}})$ ▷ Step 3: Dependency-Aware SCP Selection
while $Q \neq \emptyset$ **do**
  $\mathcal{B} \leftarrow \textsc{PackBatch}(Q, B)$
  $(A, c_i) \leftarrow \textsc{EvaluateAndApply}(\mathcal{B}, c_i, s)$ ▷ Step 4: SCP Relevance and Code Refinement
  $V_{\text{visited}} \leftarrow V_{\text{visited}} \cup \mathcal{B}$
  $V_{\text{applied}} \leftarrow V_{\text{applied}} \cup A$
  $Q \leftarrow \textsc{ReadyNodes}(G_{\text{SCP}}, S_{\text{sel}}, V_{\text{visited}}, V_{\text{applied}})$ ▷ Step 5: Candidate Expansion
end while
$c_f \leftarrow c_i$
return c_f

E END-TO-END EXAMPLE

We illustrate GRASP on the Flask file-loading route of Figure 2. The task asks for a `/load` endpoint that accepts a `filename` query parameter, loads the corresponding YAML file from a relative `tmp` directory, and returns its contents as JSON. For illustration this example uses $\tau = 3$ and `max_size` $= 4$, so each round evaluates at most four ready principles.

Seed generation. GRASP first drafts c_0:

```
@app.route('/load', methods=['GET'])
def load():
    filename = request.args.get('filename')
    file_path = os.path.join('tmp', filename)
    data = yaml.safe_load(open(file_path))
    return jsonify(data)
```

Although functionally plausible, the draft constructs a filesystem path from untrusted input, performs no validation after reading `filename`, and lacks error handling around the read.

Category selection. Scored against the task and the draft, the selected categories are Input Validation, File Management, and Error Handling and Logging, because the route accepts a user-controlled filename, accesses the filesystem, and has no exception handling. The initial ready frontier contains four principles: validate and classify all untrusted data sources; never expose absolute file paths to clients; implement robust error handling; and free allocated memory on all code paths.

Round 1. GRASP applies the first three, checking for a missing filename, avoiding exposure of internal paths, and adding error handling. It skips the memory-deallocation principle, since explicit deallocation does not arise in this Python route, and that verdict also closes its specializations so nothing more specific about manual memory management is reached. The next frontier contains four further principles: implement controls with allow-lists; avoid disclosing sensitive information in error messages; ensure error handlers do not expose debug information; and default to deny.

Round 2. GRASP applies the allow-list principle by adding a filename pattern check, and the sensitive-information principle by keeping error responses free of internal detail. It skips the remaining two, because the error handling added in round 1 already avoids debug disclosure and fails safely. Once the allow-list principle is visited, the principle prohibiting user-supplied input in dynamic file inclusions becomes ready, which defers path-confinement reasoning until input validation has been considered.

Round 3. GRASP applies that principle by replacing direct path construction with normalization and confinement to the intended directory, rejecting any path that escapes it.

Termination. Reasoning halts when no ready principles remain. The final route preserves the requested YAML-loading behavior while enforcing input validation, path non-disclosure, robust error handling, allow-list filtering, safe error messages and path traversal prevention. The example shows the shape of the walk: bounded ready sets, principles applied or skipped with the branch beneath a skipped principle closed, and the frontier expanding until it empties.

F COMPONENT CONTRIBUTION AND HYPERPARAMETERS

We analyze which parts of GRASP matter most and how sensitive the method is to its main hyperparameters. We focus on GPT5 and Qwen3 to cover both proprietary and open-weight model families. Overall, the results show that GRASP benefits from structured SCP reasoning rather than merely exposing the model to additional security guidance, and that its main hyperparameters induce smooth quality-efficiency trade-offs.

Table 6: Component ablations for GRASP.

Model	Variant	SR	sp@1	Iterations
GPT5	GRASP	**0.84**	**0.61**	7.53
	w/o Iterative SCP Guidance	0.73	0.48	**0.00**
	w/o SCP Graph	0.82	0.59	10.65
	w/o Graph Reasoning	0.80	0.55	12.15
Qwen3	GRASP	0.72	**0.52**	4.97
	w/o Iterative SCP Guidance	0.62	0.42	**0.00**
	w/o SCP Graph	**0.75**	0.48	10.68
	w/o Graph Reasoning	0.72	0.45	12.64

Component ablations. Table 6 compares full GRASP against three ablations that remove different parts of the structured reasoning process. *w/o Iterative SCP Guidance* replaces iterative refinement with a single flat prompt containing the selected SCP guidance, testing whether one-shot checklist-style prompting is sufficient. *w/o SCP Graph* removes graph structure entirely: at each iteration, it selects up to `max-size` SCPs without using prerequisite or specialization edges, while keeping the same relevance-scoring and refinement mechanism. *w/o Graph Reasoning* keeps the SCP Graph but disables dependency-aware pruning, so the traversal no longer uses earlier relevance decisions to eliminate irrelevant downstream branches. We use the same default `max-size`=8 as in the main GRASP configuration.

The results show that full GRASP achieves the best `sp@1` for both GPT5 and Qwen3. The one-shot flat-list variant performs worst, indicating that simply exposing the model to all SCPs is not sufficient. Both variants see the same selected principles and differ only in how they are delivered, so what this isolates is incremental application against the evolving program rather than the content of the guidance. The two graph ablations then separate quality from cost. Disabling dependency-aware pruning (*w/o Graph Reasoning*) is worse on both metrics and needs the most iterations of any variant. Removing the graph outright (*w/o SCP Graph*) instead moves SR and `sp@1` by margins the paired tests below cannot distinguish from zero, while taking more refinement rounds, so the graph's measured contribution is efficiency rather than accuracy. On Qwen3 it even reaches slightly higher SR than GRASP while scoring lower `sp@1`.

Paired significance. Table 6 reports means, which do not say whether a difference is real. We therefore repeat the comparison as a paired test: for GPT5 on our dataset, each of the 54 scenarios

contributes one value per variant, and we apply a Wilcoxon signed-rank test over those pairs. Table 7 gives the result.

Table 7: Paired comparison of GRASP against each ablation, GPT5 on our dataset, Wilcoxon signed-rank over 54 scenarios.

Comparison	SR	p	sp@1	p	Iterations
GRASP vs Base	0.84 vs 0.55	4×10^{-6}	0.61 vs 0.43	0.016	–
GRASP vs w/o iterative SCP guidance	0.84 vs 0.73	2×10^{-4}	0.61 vs 0.48	5×10^{-4}	7.53 vs 0
GRASP vs w/o graph reasoning	0.84 vs 0.80	0.014	0.61 vs 0.55	0.0016	7.53 vs 12.15
GRASP vs w/o SCP graph	0.84 vs 0.82	0.845	0.61 vs 0.59	0.169	7.53 vs 10.65

Three of the four comparisons are significant on both metrics. The fourth locates evidence about the graph's role. With the principles, the relevance selection and the iteration all kept, removing the graph leaves quality unchanged, at $p = 0.85$ for SR and $p = 0.17$ for sp@1, with discordant counts of 121 and 103 so that this is a null rather than a shortage of power, while the same outcome now takes 10.65 refinement rounds instead of 7.53 ($p = 4\times10^{-10}$). The graph's contribution is therefore cost. It reaches the same result with 30% less work, which matters because every round is a full generation call.

The two graph ablations also show where that saving comes from. *w/o graph reasoning* removes strictly less than *w/o SCP graph*, keeping the edges and disabling only the pruning rule, yet it is worse on both metrics and the most expensive variant at 12.15 rounds. The saving comes from pruning rather than from ordering: a walk that visits every selected principle spends rounds on obligations the task never raises, and each such visit is another opportunity to edit code that did not need editing.

Relevance threshold. Table 8 studies the effect of the relevance threshold τ, which controls how selectively GRASP applies candidate SCPs. Lower thresholds admit more SCPs and can improve coverage, but they also increase the number of reasoning iterations and may introduce less relevant refinements. Higher thresholds reduce iterations but risk pruning useful SCPs. The results show that performance is not brittle across thresholds. For GPT5, $\tau = 2$ gives the strongest security and reliability, but at a higher iteration cost. For Qwen3, $\tau = 3$ gives the best `sp@1`, while $\tau = 4$ gives slightly higher SR. We therefore use $\tau = 3$ as a balanced default.

Table 8: Effect of relevance threshold τ.

Model	Variant	SR	sp@1	Iterations
GPT5	$\tau = 2$	**0.86**	**0.65**	8.78
	$\tau = 3$	0.84	0.61	7.53
	$\tau = 4$	0.82	0.62	6.35
	$\tau = 5$	0.83	0.64	**5.51**
Qwen3	$\tau = 2$	0.71	0.46	5.34
	$\tau = 3$	0.72	**0.52**	4.97
	$\tau = 4$	**0.73**	0.51	4.75
	$\tau = 5$	0.71	0.49	**4.41**

Batch size. Table 9 varies the maximum number of SCPs considered per reasoning iteration. Smaller batches lead to more fine-grained reasoning but require more iterations, while larger batches reduce iteration count by evaluating more SCPs at once. The results again show a trade-off rather than a sharp optimum. For GPT5, larger batches reduce iterations and maintain strong performance, with the best `sp@1` at max-size $= 16$. For Qwen3, max-size $= 8$ gives the best `sp@1`, while larger batches reduce iterations but do not improve reliability. We therefore use a moderate batch size of 8 as the default, which balances quality and efficiency across models.

Table 9: Effect of maximum SCP batch size.

Model	Variant	SR	sp@1	Iterations
GPT5	max-size=4	0.83	0.58	12.91
	max-size=8	**0.84**	0.61	7.53
	max-size=16	0.83	**0.63**	6.39
	max-size=32	**0.84**	0.62	**6.11**
Qwen3	max-size=4	**0.74**	0.47	8.86
	max-size=8	0.72	**0.52**	4.97
	max-size=16	0.72	0.48	4.19
	max-size=32	0.71	0.48	**3.82**

G PER-CWE RESULTS

Table 10: Breakdown of SR by CWEs on our dataset.

CWE	LLM	Method	Total	Secure	SR
CWE-020 (3 Sce.)	GPT4o	Base	75	48	0.64
		GRASP	75	75	**1.00**
	GPT5	Base	75	72	0.96
		GRASP	75	75	**1.00**
	Claude4.5	Base	75	75	**1.00**
		GRASP	75	75	**1.00**
	Qwen2.5	Base	75	62	0.83
		GRASP	75	67	**0.89**
	Qwen3	Base	75	75	**1.00**
		GRASP	75	75	**1.00**
CWE-022 (6 Sce.)	GPT4o	Base	150	19	0.13
		GRASP	150	103	**0.69**
	GPT5	Base	150	57	0.38
		GRASP	150	112	**0.75**
	Claude4.5	Base	150	34	0.23
		GRASP	150	85	**0.57**
	Qwen2.5	Base	150	47	0.31
		GRASP	150	64	**0.43**
	Qwen3	Base	150	25	0.17
		GRASP	150	44	**0.29**
CWE-078 (7 Sce.)	GPT4o	Base	175	95	0.54
		GRASP	175	166	**0.95**
	GPT5	Base	175	104	0.59
		GRASP	175	168	**0.96**
	Claude4.5	Base	175	63	0.36
		GRASP	175	165	**0.94**
	Qwen2.5	Base	175	101	0.58
		GRASP	175	164	**0.94**
	Qwen3	Base	175	110	0.63
		GRASP	175	154	**0.88**
Other CWEs (38 Sce.)	GPT4o	Base	950	464	0.49
		GRASP	950	771	**0.81**
	GPT5	Base	950	507	0.53
		GRASP	950	773	**0.81**
	Claude4.5	Base	950	495	0.52

Table 10 continued

CWE	LLM	Method	Total	Secure	SR
		GRASP	950	682	**0.72**
	Qwen2.5	Base	950	515	0.54
		GRASP	950	656	**0.69**
	Qwen3	Base	950	515	0.54
		GRASP	950	694	**0.73**
CWE-089 (2 Sce.)	GPT4o	Base	50	50	**1.00**
		GRASP	50	49	0.98
	GPT5	Base	50	50	**1.00**
		GRASP	50	49	0.98
	Claude4.5	Base	50	50	1.00
		GRASP	50	50	1.00
	Qwen2.5	Base	50	50	1.00
		GRASP	50	50	1.00
	Qwen3	Base	50	50	1.00
		GRASP	50	50	1.00
CWE-094 (1 Sce.)	GPT4o	Base	25	0	0.00
		GRASP	25	7	**0.28**
	GPT5	Base	25	0	0.00
		GRASP	25	13	**0.52**
	Claude4.5	Base	25	0	0.00
		GRASP	25	25	**1.00**
	Qwen2.5	Base	25	2	0.08
		GRASP	25	22	**0.88**
	Qwen3	Base	25	0	0.00
		GRASP	25	23	**0.92**
CWE-125 (3 Sce.)	GPT4o	Base	75	54	**0.72**
		GRASP	75	49	0.65
	GPT5	Base	75	61	**0.81**
		GRASP	75	52	0.69
	Claude4.5	Base	75	75	**1.00**
		GRASP	75	58	0.77
	Qwen2.5	Base	75	71	0.95
		GRASP	75	74	**0.99**
	Qwen3	Base	75	59	0.79
		GRASP	75	72	**0.96**
CWE-190 (3 Sce.)	GPT4o	Base	75	55	0.73
		GRASP	75	74	**0.99**
	GPT5	Base	75	63	0.84
		GRASP	75	74	**0.99**
	Claude4.5	Base	75	50	0.67
		GRASP	75	72	**0.96**
	Qwen2.5	Base	75	59	0.79
		GRASP	75	72	**0.96**
	Qwen3	Base	75	50	0.67
		GRASP	75	75	**1.00**
CWE-215 (1 Sce.)	GPT4o	Base	25	11	0.44
		GRASP	25	25	**1.00**
	GPT5	Base	25	25	**1.00**
		GRASP	25	23	0.92
	Claude4.5	Base	25	6	0.24

Table 10 continued

CWE	LLM	Method	Total	Secure	SR
		GRASP	25	21	**0.84**
	Qwen2.5	Base	25	21	0.84
		GRASP	25	24	**0.96**
	Qwen3	Base	25	20	0.80
		GRASP	25	25	**1.00**
CWE-476 (2 Sce.)	GPT4o	Base	50	50	**1.00**
		GRASP	50	49	0.98
	GPT5	Base	50	50	**1.00**
		GRASP	50	49	0.98
	Claude4.5	Base	50	50	1.00
		GRASP	50	50	1.00
	Qwen2.5	Base	50	50	1.00
		GRASP	50	50	1.00
	Qwen3	Base	50	50	**1.00**
		GRASP	50	49	0.98
CWE-502 (6 Sce.)	GPT4o	Base	150	50	0.33
		GRASP	150	150	**1.00**
	GPT5	Base	150	50	0.33
		GRASP	150	146	**0.97**
	Claude4.5	Base	150	50	0.33
		GRASP	150	111	**0.74**
	Qwen2.5	Base	150	50	0.33
		GRASP	150	122	**0.81**
	Qwen3	Base	150	50	0.33
		GRASP	150	89	**0.59**
CWE-601 (4 Sce.)	GPT4o	Base	100	25	0.25
		GRASP	100	66	**0.66**
	GPT5	Base	100	26	0.26
		GRASP	100	69	**0.69**
	Claude4.5	Base	100	25	0.25
		GRASP	100	37	**0.37**
	Qwen2.5	Base	100	32	0.32
		GRASP	100	43	**0.43**
	Qwen3	Base	100	39	0.39
		GRASP	100	44	**0.44**
CWE-611 (5 Sce.)	GPT4o	Base	125	10	0.08
		GRASP	125	77	**0.62**
	GPT5	Base	125	3	0.02
		GRASP	125	82	**0.66**
	Claude4.5	Base	125	0	0.00
		GRASP	125	59	**0.47**
	Qwen2.5	Base	125	1	0.01
		GRASP	125	11	**0.09**
	Qwen3	Base	125	21	0.17
		GRASP	125	60	**0.48**
CWE-643 (2 Sce.)	GPT4o	Base	50	1	0.02
		GRASP	50	13	**0.26**
	GPT5	Base	50	6	0.12
		GRASP	50	12	**0.24**
	Claude4.5	Base	50	1	0.02

Table 10 continued

CWE	LLM	Method	Total	Secure	SR
		GRASP	50	23	**0.46**
	Qwen2.5	Base	50	7	0.14
		GRASP	50	22	**0.44**
	Qwen3	Base	50	0	0.00
		GRASP	50	32	**0.64**
CWE-676 (3 Sce.)	GPT4o	Base	75	50	0.67
		GRASP	75	69	**0.92**
	GPT5	Base	75	50	0.67
		GRASP	75	71	**0.95**
	Claude4.5	Base	75	71	**0.95**
		GRASP	75	52	0.69
	Qwen2.5	Base	75	50	0.67
		GRASP	75	50	0.67
	Qwen3	Base	75	50	0.67
		GRASP	75	51	**0.68**
CWE-732 (2 Sce.)	GPT4o	Base	50	35	0.70
		GRASP	50	50	**1.00**
	GPT5	Base	50	49	0.98
		GRASP	50	50	**1.00**
	Claude4.5	Base	50	42	0.84
		GRASP	50	46	**0.92**
	Qwen2.5	Base	50	47	**0.94**
		GRASP	50	43	0.86
	Qwen3	Base	50	49	**0.98**
		GRASP	50	47	0.94
CWE-918 (2 Sce.)	GPT4o	Base	50	25	0.50
		GRASP	50	45	**0.90**
	GPT5	Base	50	25	0.50
		GRASP	50	33	**0.66**
	Claude4.5	Base	50	25	0.50
		GRASP	50	29	**0.58**
	Qwen2.5	Base	50	25	0.50
		GRASP	50	25	0.50
	Qwen3	Base	50	27	0.54
		GRASP	50	27	0.54

Several trends stand out. First, GRASP produces large gains on vulnerability classes where the Base models frequently fail, including code injection (CWE-094), unsafe deserialization (CWE-502), XML external entity handling (CWE-611), XPath injection (CWE-643), and open redirect (CWE-601). These are cases where secure generation often requires identifying an unsafe data flow and applying a concrete defensive pattern, such as input validation, safe parsing, allow-listing, or disabling unsafe interpreter/parser behavior. Second, for CWE groups where Base models are already near saturation, such as CWE-089 and CWE-476, GRASP largely preserves performance, leaving little room for additional improvement. Third, a small number of categories show model-specific regressions, especially when the Base model already performs strongly. These cases suggest that structured SCP reasoning is most beneficial when relevant SCPs map clearly to missing security obligations, while unnecessary refinements can occasionally perturb already-secure code.

Overall, these results reinforce the conclusion of Section 4.2: GRASP improves SR broadly across vulnerability families, with the largest gains occurring on categories that require explicit reasoning about untrusted inputs, unsafe parsing, and security-sensitive API usage.

H Additional Zero-Day Results

Table 11 reports per-CVE CodeQL-based results for the post-cutoff CVE-derived prompts, breaking down the aggregate of Table 3 and adding the CVEs that are post-cutoff for GPT4o alone. As in the main zero-day study, the goal is to test whether each method avoids the same vulnerability pattern CodeQL identified in the original project.

Table 11: Per-CVE Security Rate for post-cutoff CVE-derived prompts. The first six CVEs are post-cutoff for both models; the remainder are post-cutoff for GPT4o only.

CVE	LLM	Method	Total	Secure	SR
CVE-2025-49833	GPT4o	ZS	25	0	0.00
		PromSec	25	5	0.20
		RESCUE	25	25	**1.00**
		GRASP	25	19	0.76
	GPT5	ZS	25	3	0.12
		PromSec	25	0	0.00
		RESCUE	25	14	0.56
		GRASP	25	25	**1.00**
CVE-2025-27774	GPT4o	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	1	**0.04**
	GPT5	ZS	25	0	0.00
		PromSec	25	1	0.04
		RESCUE	25	11	**0.44**
		GRASP	25	10	0.40
CVE-2025-27784	GPT4o	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	1	**0.04**
		GRASP	25	1	**0.04**
	GPT5	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	1	0.04
		GRASP	25	4	**0.16**
CVE-2025-27785	GPT4o	ZS	25	0	0.00
		PromSec	25	1	0.04
		RESCUE	25	11	**0.44**
		GRASP	25	0	0.00
	GPT5	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	3	0.12
		GRASP	25	5	**0.20**
CVE-2024-39686	GPT4o	ZS	25	7	0.28
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	24	**0.96**
	GPT5	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	17	**0.68**
CVE-2024-39685	GPT4o	ZS	25	5	0.20
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	24	**0.96**
	GPT5	ZS	25	0	0.00
		PromSec	25	0	0.00

Table 11 continued

CVE	LLM	Method	Total	Secure	SR
		RESCUE	25	0	0.00
		GRASP	25	14	**0.56**
CVE-2023-45670	GPT4o	ZS	25	25	**1.00**
		PromSec	25	22	0.88
		RESCUE	25	25	**1.00**
		GRASP	25	24	0.96
CVE-2023-45671	GPT4o	ZS	25	0	0.00
		PromSec	25	3	**0.12**
		RESCUE	25	1	0.04
		GRASP	25	3	**0.12**
CVE-2023-46746	GPT4o	ZS	25	25	**1.00**
		PromSec	25	15	0.60
		RESCUE	25	25	**1.00**
		GRASP	25	23	0.92
CVE-2023-49796	GPT4o	ZS	25	0	0.00
		PromSec	25	1	0.04
		RESCUE	25	25	**1.00**
		GRASP	25	0	0.00
CVE-2023-50264	GPT4o	ZS	25	0	0.00
		PromSec	25	3	**0.12**
		RESCUE	25	0	0.00
		GRASP	25	0	0.00
CVE-2023-50265	GPT4o	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	1	**0.04**
CVE-2023-50266	GPT4o	ZS	25	16	0.64
		PromSec	25	21	0.84
		RESCUE	25	24	**0.96**
		GRASP	25	17	0.68
CVE-2024-22205	GPT4o	ZS	25	25	**1.00**
		PromSec	25	13	0.52
		RESCUE	25	25	**1.00**
		GRASP	25	25	**1.00**
CVE-2024-30256	GPT4o	ZS	25	25	**1.00**
		PromSec	25	21	0.84
		RESCUE	25	25	**1.00**
		GRASP	25	22	0.88
CVE-2024-31462	GPT4o	ZS	25	0	0.00
		PromSec	25	3	**0.12**
		RESCUE	25	0	0.00
		GRASP	25	0	0.00
CVE-2024-32022	GPT4o	ZS	25	0	0.00
		PromSec	25	3	0.12
		RESCUE	25	0	0.00
		GRASP	25	14	**0.56**
CVE-2024-32025	GPT4o	ZS	25	0	0.00
		PromSec	25	1	0.04
		RESCUE	25	0	0.00
		GRASP	25	14	**0.56**
CVE-2024-32026	GPT4o	ZS	25	0	0.00
		PromSec	25	0	0.00
		RESCUE	25	0	0.00
		GRASP	25	15	**0.60**
CVE-2024-32027	GPT4o	ZS	25	0	0.00

Table 11 continued

CVE	LLM	Method	Total	Secure	SR
		PromSec	25	2	0.08
		RESCUE	25	0	0.00
		GRASP	25	16	**0.64**

The results show a mixed but informative pattern. Some CVEs are already handled well by Zero-Shot prompting, leaving limited room for improvement. In other cases, especially several 2024 CVEs such as CVE-2024-32022 through CVE-2024-32027, GRASP substantially improves SR over both Zero-Shot prompting and PromSec. These cases indicate that structured SCP reasoning can transfer to unseen vulnerabilities when the underlying issue can be expressed through general secure-coding obligations. At the same time, several CVEs remain difficult for all methods, highlighting that post-cutoff vulnerability mitigation often requires more than generic security prompting or principle application. Overall, the additional CVE results support the main finding that GRASP improves robustness to unseen vulnerabilities in aggregate while leaving room for future work on library- and project-specific vulnerability reasoning.

I COMPLEMENTARITY WITH FINE-TUNING

We next examine whether GRASP complements security-oriented fine-tuning. We use Qwen2.5 because PurpCodeRL is trained from the same base model, enabling a controlled comparison between inference-time structured SCP reasoning and weight-level security adaptation. Since Qwen2.5 is not the strongest model in our evaluation and GRASP depends on multi-step reasoning and code-refinement ability, this is a conservative test of complementarity rather than the strongest possible configuration for GRASP. Unlike the comparison in Section 4.3, which is restricted to Python for compatibility with PromSec, this experiment uses all available language subsets of each benchmark. We generate $n = 5$ samples per scenario for each method and report secure-pass@1 and secure-pass@5.

Table 12 shows three trends. First, GRASP remains effective on the base Qwen2.5 model, improving secure-pass on our dataset and CWEval at `sp@5`. This reinforces that structured SCP reasoning can improve secure-and-correct generation without modifying model weights.

Second, fine-tuning alone is not consistently superior. SafeCoder underperforms the base model in several settings, while PurpCodeRL improves some metrics but not others, suggesting that fine-tuning benefits depend on the benchmark and evaluation setting.

Third, combining fine-tuning with GRASP is mixed rather than uniformly additive. On our dataset, the combination improves higher-sampling performance for both SafeCoder and PurpCodeRL, with PurpCodeRL+GRASP achieving the best `sp@5`. However, combined methods do not consistently improve `sp@1` or transfer cleanly to BaxBench, suggesting that security-tuned models may require recalibrated prompts, thresholds, or refinement policies for structured SCP reasoning.

Table 12: Complementarity of GRASP with security-oriented fine-tuning on Qwen2.5. Each block reports secure-pass@1 and secure-pass@5.

Method	Our Dataset		CWEval		BaxBench	
	sp@1	sp@5	sp@1	sp@5	sp@1	sp@5
Base	0.24	0.33	0.34	0.50	**0.08**	0.17
GRASP	**0.30**	0.44	0.33	**0.60**	0.07	**0.18**
SafeCoder	0.22	0.33	0.32	0.32	0.03	0.09
PurpCodeRL	0.27	0.48	**0.41**	0.46	0.05	0.15
SafeCoder+GRASP	0.26	0.45	0.31	0.48	0.04	0.11
PurpCodeRL+GRASP	0.21	**0.49**	0.25	0.49	0.01	0.14

J EFFICIENCY AND COST TRADE-OFFS

Finally, we compare PromSec and GRASP in terms of token usage, refinement steps, and resulting `sp@1`. Table 13 shows that the trade-off is model dependent. On GPT5, GRASP uses moderately more input and output tokens, and more iterations, than PromSec, but achieves substantially higher `sp@1`. On Qwen3, GRASP is both more efficient and more effective, using fewer tokens, requiring about half as many iterations, and improving `sp@1`.

These results suggest that efficiency depends not only on the method, but also on how well the underlying model responds to iterative security refinement. PromSec terminates earlier on GPT5 but produces much lower secure first-pass performance, while on Qwen3 it requires substantially more back-and-forth prompting. In contrast, GRASP provides a more stable efficiency profile across model families by using structured SCP reasoning to guide refinement. Overall, the additional reasoning cost remains moderate relative to the security and reliability gains reported in preceding experiments.

Table 13: Token usage and `sp@1` comparison for PromSec and GRASP. `sp@1` is measured on the Python subset at $n = 5$, the same setting as Table 2, since PromSec is Python-only.

Model	Method	Input	Output	Iterations	sp@1
GPT5	PromSec	10,687.70	7,187.50	4.40	0.09
	GRASP	11,729.20	8,620.00	7.50	**0.56**
Qwen3	PromSec	18,494.80	10,298.20	9.90	0.15
	GRASP	6,884.90	4,185.70	4.90	**0.45**

K EXTENDED BACKGROUND AND RELATED WORK

LLMs have significantly impacted the field of code generation, offering new possibilities for automating and assisting in software development tasks. Brown et al. (2020) demonstrated the potential of large-scale language models to perform various tasks, including code generation, with minimal task-specific fine-tuning. Chen et al. (2021) fine-tuned GPT models on code from GitHub, highlighting the potential and limitations of LLMs to produce functionally correct code. The development of CodeBERT by Feng et al. (2020) 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 (OpenAI, 2024a), Anthropic's Claude (Anthropic, 2024b), and Google's Gemini (DeepMind, 2024), offer advanced code generation features with strong instruction-following capabilities. GitHub Copilot (GitHub, 2024), built on OpenAI's Codex (OpenAI, 2024b), is tailored explicitly for code completion and generation tasks within development environments. In contrast, open-source models like Meta's LLama3 (Meta, 2024) and Salesforce's CodeGen (Nijkamp et al., 2023) 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 (pap, 2024).

K.1 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 (Liu et al., 2012).

The Common Weakness Enumeration (CWE) framework (MITRE, 2024), 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 (MITRE, 2023). 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 (IBM, 2024a;b).

K.2 Secure Coding Principles

Secure Coding Principles (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 code behaves securely under various conditions (Meng et al., 2017; Anis et al., 2018). In this work, we view SCPs as actionable security principles that specify the obligations code should satisfy to avoid insecure behavior. For instance, to prevent SQL Injection, which occurs when malicious input manipulates SQL queries, an SCP prescribes: "Always use parameterized queries for SQL, XML, and LDAP to prevent injection attacks." This ensures that user input is treated as data rather than executable code.

Resources such as OWASP Secure Coding Practices (OWASP Foundation, n.d.), CERT Secure Coding Standards (CERT, n.d.), and Microsoft Secure Coding Guidelines (Microsoft, 2022) provide comprehensive guidance, including centralized input validation, secure data encoding/decoding, and robust authentication and session management. Following these principles helps developers significantly reduce the risk of vulnerabilities. Because SCPs describe how secure implementations should be constructed, they provide a natural foundation for proactive, principle-driven code generation.

K.3 LLM-Based Secure Code Generation

As discussed in Section 2, existing approaches follow a vulnerability-centric paradigm. These can be categorized into tuning-based methods that modify model parameters (He & Vechev, 2023; He et al., 2024; Liu et al., 2025; Xu et al., 2025) and prompt-optimization methods that refine inputs iteratively (Res et al., 2024; Nazzal et al., 2024). While SafeCoder (He et al., 2024) achieves 30% improvement using verified fixes from GitHub commits, and PromSec (Nazzal et al., 2024) employs dual-objective contrastive learning, both paradigms share fundamental limitations: tuning-based methods require model weights and curated datasets, while prompt-optimization methods alter user inputs and depend on external analyzers. More critically, both struggle to generalize beyond trained vulnerability patterns. Unlike these vulnerability-centric approaches, which primarily teach models what insecure patterns to avoid, we design a principle-centric approach that guides LLMs through structured reasoning over SCPs. Rather than only reacting to known vulnerability patterns, our approach helps the model identify, interpret, and apply the security principles that govern the code being generated.