The Journal of Systems and Software 232 (2026) 112682

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Discrete prompt optimization using genetic algorithm for secure Python code generation

Catherine Tony [a,*], Maura Pintor [b], Max Kretschmann [a], Riccardo Scandariato [a]

[a] Hamburg University of Technology, Germany
[b] University of Cagliary, Italy

ARTICLE INFO

Professor Yan Cai

Keywords:
LLMs
Secure code generation
Prompt optimization
Genetic algorithms

ABSTRACT

Large language models (LLMs) have become powerful tools that enable novice developers to generate production-level code. However, research has highlighted the security risks associated with such code generation, due to the high volume of generated software vulnerabilities. Recent studies have explored various techniques for automatically optimizing prompts to elicit desired responses from LLMs. Among these methods, Genetic Algorithms (GAs), which search for optimal solutions by evolving an initial population of candidates through iterative mutations, have gained attention as a lightweight and effective prompt optimization approach that does not require large datasets or access to model weights. However, their potential has not yet been examined in the context of secure code generation. In this paper, we use GA to develop a discrete prompt optimization pipeline specifically designed for secure code generation. We introduce two domain-specific prompt mutation techniques and assess how incorporating these security-focused mutations alongside general-purpose techniques, such as back translation and paraphrasing, affects the security of Python code generated by LLMs. Results demonstrate that our security-specific mutation techniques led to prompts with richer security context compared to the generic mutation techniques. Furthermore, combining these techniques with generic mutations substantially reduced the number of security weaknesses in the LLM-generated code. We also observed that prompts optimized for a particular LLM tend to perform best on that same model, highlighting the importance of model-specific prompt optimization.

1. Introduction

Large Language Models (LLMs) have become powerful tools to process and generate natural language (NL) text. Recently, LLMs have been studied for code generation tasks, leading to powerful tools to accelerate the work done by developers. However, there is a tangible risk when using these models for fast-paced code development, especially because these models perform poorly in generating secure code and introduce even more vulnerabilities than usual, due to the memorization of vulnerable code from their training data (Pearce et al., 2022; Siddiq et al., 2022). For this reason, exploring secure code generation using LLMs has become a critical area of research. Studies indicate that structuring prompts with specific patterns and words improves the responses from LLMs, a technique called prompt engineering. By refining the prompts using this approach, the outcomes of code generation can potentially be enhanced in terms of security.

Several papers have explored various *manually* created prompting patterns with the aim of identifying optimal strategies for coding tasks, like self-planning, self-refine, and so on (Madaan et al., 2023; Jiang et al., 2023; White et al., 2023). However, these efforts are only providing generic guidance, as pinpointing the specific prompt variation that yields optimal results for different types of tasks is a complex process that cannot be realistically performed manually. This is where *automated* prompt optimization becomes crucial. Prompt optimization can be viewed as a search problem across a large space of prompt variants to identify the most effective ones for generating optimal outputs, in our case, secure code. Exhaustively evaluating each variation, however, is computationally expensive. Recent advances have introduced methods like prompt tuning (Lester et al., 2021) and black-box tuning (Sun et al., 2022b), which automate prompt optimization across various natural language processing tasks. While these approaches are less demanding than full model fine-tuning, they still rely on large amounts of labeled data and substantial computational resources to be effective. Additionally, a majority of them work with continuous soft prompts, i.e, prompts in the form of machine-readable vector embeddings, making them difficult to deploy on closed-source models like GPT-4 and Claude.

* Corresponding author.
E-mail addresses: catherine.tony@tuhh.de (C. Tony), maura.pintor@unica.it (M. Pintor), max.kretschmann@tuhh.de (M. Kretschmann), riccardo.scandariato@tuhh.de (R. Scandariato).

https://doi.org/10.1016/j.jss.2025.112682
Received 2 July 2025; Received in revised form 16 September 2025; Accepted 20 October 2025
Available online 26 October 2025

Recently, Genetic Algorithms (GAs), which work by evolving a population of candidate solutions through different mutation processes, have gained popularity in the field of prompt optimization due to their simplicity and ability to efficiently explore large and complex search spaces. Existing works have employed GAs for prompt optimizations for tasks such as sentence completion, word sense disambiguation, and classification (Prasad et al., 2023; Xu et al., 2022; Zhao et al., 2023), where they optimize prompts by generating new candidates through mutation techniques such as back translation (Xu et al., 2022) and paraphrasing (Prasad et al., 2023). However, no work has explored the adaptation of GAs for the secure code generation domain. It would be particularly interesting to investigate whether generic mutation techniques are sufficient or if the approach would benefit from more domain-specific mutation techniques tailored to code security.

In this paper, we adapt GA to optimize discrete (human-readable) prompts for enhancing the security of LLM-generated code. Specifically, we examine the effect of incorporating security-specific prompt mutation techniques, alongside generic ones, on the security of the generated code. To achieve this, we developed an optimization pipeline that includes prompt scoring and mutation techniques tailored for secure code generation. We introduce two security-specific, LLM-assisted prompt mutation techniques, *self-guided* and *feedback-guided*, and evaluate their impact on the security of code generated using 5 popular LLMs. We focused on code generated in the Python programming language due to its continuing popularity among developers for different software development use cases (PYPL Index, 2024; TIOBE Index, 2024).

Our findings indicate that generic prompt mutation techniques have limited capacity to produce prompts enriched with meaningful security context. In contrast, the security-specific mutation techniques generated more sophisticated prompts that embedded stronger security cues. As a result, incorporating security-specific mutation techniques in the prompt optimization pipeline led to a further reduction in security weaknesses in LLM-generated code compared to using only generic techniques. Moreover, the implemented optimization pipeline, which relied on just 36 coding reference tasks, was able to produce prompts that generalized well to the unseen coding tasks from a different dataset when using the security-specific mutation techniques. Finally, we investigated the transferability of prompts optimized on an LLM to other models, which would open the possibility to optimize locally (e.g., on publicly-available open-source models) and use the same prompts on proprietary models (such as GPT-4). However, we observed that the optimized prompts perform best when used with the same LLM on which they were originally optimized. The main contributions of this work can be summarized as follows:

- we present a prompt optimization approach powered by genetic algorithms for secure code generation;
- we propose 2 new security-specific LLM-based prompt mutation techniques;
- we compare the impact of adding security-specific prompt mutation techniques to generic techniques on LLM-generated code security;
- we investigate the generalizability of the optimized prompts to new datasets containing previously unseen types of coding tasks;
- we study the transferability and portability of the optimized prompts across LLMs.

The rest of this paper is organized as follows. We start our analysis by discussing the related work in Section 2, then introduce our pipeline and experimental evaluation in Section 3. Section 5 presents the results, and Section 6 provides additional discussion of the findings. This is followed by limitations (Section 7), conclusions (Section 8) and information on the replication package (Section 9).

2. Related work

In this section, we review studies that examine the security of LLM-generated code, followed by an overview of existing prompt optimization techniques, including those based on GAs. We then present approaches for enhancing the security of LLM-generated code, with a special focus on prompt-based methods.

2.1. Security of LLM-generated code

Several studies have examined the presence of security weaknesses in code generated by LLMs. Pearce et al. (2022) evaluated C and Python code generated by GitHub Copilot across 54 high-risk security scenarios, finding that 40 % of the code completions contained security vulnerabilities. Jesse et al. (2023) analyzed the prevalence of "simple, stupid bugs" (SStuBs) in code produced by Codex and other LLMs, discovering that these models generated twice as many SStuBs as correct code. Similarly, Perry et al. (2023) conducted a study with 47 developers using a Codex-powered AI assistant to complete five security-related programming tasks in Python, JavaScript, and C. Their findings showed that developers assisted by the AI were more likely to produce insecure code in four out of the five tasks. Another study (Khoury et al., 2023) evaluated code generated by ChatGPT, powered by the GPT-3.5 series, for 21 security-sensitive coding tasks across five programming languages, including C/C++, Python, and Java. They found that the generated code consistently fell below minimum security standards. Similarly, Siddiq et al. (2022) investigated the impact of code and security smells present in the training data on the outputs of transformer-based models like GitHub Copilot. Their findings showed that these models tended to leak smells and non-standard coding practices from their training data into the generated code.

2.2. Prompt optimization

Given the significant impact prompts have on eliciting desirable outputs from LLMs, prompt optimization is an actively researched area. Prompt optimization can be divided into two categories based on the type of prompts they deal with: (i) *soft prompt optimization* and (ii) discrete prompt optimization.

2.2.1. Soft prompt optimization

In this kind of optimization, soft prompts which are in the form of continuous vector embeddings are optimized to improve the LLM responses. Prompt tuning (Lester et al., 2021) an optimization technique that learns soft prompts to condition language models to produce desirable results. Wang et al. (2022a) investigated prompt tuning for code intelligence tasks and observed that prompt tuning outperforms fine-tuning for pre-trained models of smaller sizes. Gu et al. (2022) proposed a framework called Pre-trained Prompt Tuning (PPT), where the initial prompt tokens are pre-trained using self-supervised classification tasks on large, unlabeled data corpora. These pre-trained prompt tokens are then used for prompt tuning on downstream tasks. Evaluations conducted on multiple datasets containing several types of natural language classification tasks showed that it significantly improved the results of prompt tuning. Similarly, another framework called Unified Prompt Tuning (UPT) (Wang et al., 2022b) employs a unified framework that leverages prompting knowledge learned from non-target NLP datasets to improve performance on target tasks. Soft Prompt Tuning (SPoT) was proposed by Vu et al. (2022) that uses an intermediate training stage between language model pre-training and target prompt tuning, where a prompt is learned on one or more source tasks. This learned prompt is then used to initialize the prompt for the target task. They evaluated the approach on 26 natural language processing tasks and reported improvement over prompt tuning. In addition to the above works, multi-task prompt tuning (Wang et al., 2023) and p-tuning (Liu et al., 2023) are also notable works in this area. Although the above approaches show promise for enhancing the model performance, they perform gradient-based optimization that requires pre-trained model weights which are not always readily available. Additionally, these methods are not

applicable to models that do not allow token embeddings, such as the GPT-series models.

Black-Box Tuning (BBT) is another form of tuning approach proposed by Sun et al. (2022b) to optimize prompts without requiring direct access to the model's gradients or internal parameters. In this approach, the continuous prompt prepended to the input text is optimized by iteratively invoking the model inference API. Applied across various natural language understanding tasks, this method showcased enhancements over traditional prompt tuning. In a follow-up work, Sun et al. (2022a) introduced BBTv2, which works by prepending continuous prompts to every layer of the pre-trained model, employing a divide-and-conquer gradient-free algorithm to optimize these prompts. BBTv2 achieved performance comparable to full model tuning while surpassing the results of both prompt tuning and the original BBT. Zheng et al. (2024b) proposed the idea that nearly optimal prompts for tasks with similar characteristics reside within a shared subspace. They hypothesized that prompts optimized within the subspace of source tasks could yield improved results when applied to a target task sharing similarities with the source tasks. Building on this concept, they introduced Black-box Tuning with Subspace Learning (BSL). Han et al. (2023) employed gradient descent with derivative-free optimization (GDFO) through knowledge distillation for task-specific continuous prompt optimization. They reported improved results over BBT and BBTv2 as well as prompt tuning. Despite utilizing optimizations that do not rely on model weights, the above-mentioned black-box tuning techniques focus on optimizing continuous tokens, which may not be compatible with models that do not support token embeddings, as mentioned earlier. In addition to the above prompt and black-box tuning, other tuning techniques that are worth mentioning include *prefix tuning* (Li and Liang, 2021; Qian et al., 2022), *instruction tuning* (Xu et al., 2023; Gupta et al., 2022), and *reinforcement learning* (Deng et al., 2022).

2.2.2. Discrete prompt optimization

Discrete prompt optimization is the process of systematically searching for and refining prompts composed of discrete natural language or human-readable tokens. Several approaches have been proposed that attempt to optimize discrete prompts. For instance, Cheng et al. (2024) introduced a method called Black-Box Prompt Optimization (BPO) that uses an LLM to optimize prompts to perform NLP tasks. Their method begins with a labeled dataset consisting of prompts paired with both good and bad responses. This dataset is provided to an LLM, which generates optimized prompts that align more closely with the good responses, thus generating a new dataset containing original prompts and their optimized counterparts. The new dataset was used to train a sequence-to-sequence model capable of automatically generating optimized prompts. Aside from this work, we observed that a large majority of discrete prompt optimization approaches are based on **Genetic Algorithms (GAs)** or evolutionary algorithms in general. These algorithms are designed to discover optimal candidate solutions from a search space through the process of iterative refinement and selection that mimics the process of natural selection (Mitchell, 1998). Through appropriate modification or *mutation* techniques, GA iteratively evolves a population of potential solutions, favoring those with desirable traits or characteristics. Gradient-free Instructional Prompt Search (GRIPS), introduced by Prasad et al. (2023), utilizes GA to optimize instructional prompts. GRIPS begins with a manually crafted seed prompt and iteratively modifies instructions using operations like delete, swap, paraphrase, and addition. These variations are evaluated on a small sample of cases to greedily search for optimal prompts. Evaluation on the "Natural Instructions" dataset (Mishra et al., 2022) for general-purpose natural language classification tasks showed superior effectiveness compared to manual rewriting and exemplar prompt search. In contrast, Genetic Prompt Search (GPS) by Xu et al. (2022) also employs GA but generates prompt variations differently, using techniques like back translation, cloze, and sentence continuation. This approach is evaluated on multiple datasets containing different NLP tasks such as natural language inference (Nie et al., 2020), coreference resolution (Sakaguchi et al., 2021), sentence completion (Zellers et al., 2019; Roemmele et al., 2011) and word sense disambiguation (Pilehvar and Camacho-Collados, 2019). These evaluations showed that GPS performed better than GRIPS. EvoPrompt (Guo et al., 2024) is another work that is akin to GPS and GRIPS that uses GA to optimize prompts. EvoPrompt utilizes evolutionary techniques like crossover and mutation to generate prompt candidates. Genetic Algorithm for Predictive Probability guided Prompting (GAP3) (Zhao et al., 2023) is another GA-based approach that optimizes prompts using predictive probabilities to select tokens for prompt mutation. Starting with an empty prompt template, it iteratively constructs prompt parts through probabilistic mutation based on a labeled training set. Evaluation across 7 benchmarks (Sun et al., 2022a) comprising different NLP tasks exhibited improvement over GPS and GRIPS methodologies. All the above approaches were developed for general natural language processing tasks. In contrast, Ye et al. (2025) recently introduced ProChemy, a method specifically designed to refine prompts for code generation. Their approach mutates prompts by leveraging an LLM to generate variations through rephrasing, improving clarity, or altering structural presentation. The resulting prompts are then evaluated based on the quality of the code generated from them. While this method bears similarity to ours, the prompt mutation techniques nor the scoring approach account for code security considerations.

Other works that employ concepts similar to a GA include the study by Jiang et al. (2020), where they optimized prompts for knowledge extraction from LLMs through a systematic exploration of the prompt space. They achieve this by generating diverse variations of prompts using techniques such as mining and paraphrasing, aiming to identify the most effective prompts. On the other hand, Hao et al. (2023) used paraphrasing to harvest a massive knowledge graph of arbitrary relations from LLMs. Pryzant et al. (2023) used textual gradients in the form of model feedback on the performance of the prompt on a sample set of tasks to optimize prompts while employing beam search and bandit selection to guide the process. Automatic Prompt Engineer (APE) (Zhou et al., 2023) is another method that optimizes a prompt or an instruction by searching over a pool of instruction candidates generated by an LLM and selects the most suitable candidate based on computed evaluation scores. An approach called OPRO was proposed by Yang et al. (2024) that performs an iterative search guided by LLMs to find optimal prompts. Here, an optimizer LLM receives a meta-prompt containing candidate prompts with their scores, task input-output examples, and instructions to generate improved candidates. These new prompts are scored by a separate scorer LLM and fed back into the optimizer in the next iteration, repeating until an exit criterion is reached.

While a few of the above presented GA-based approaches, such as ProChemy, utilize LLM-guided mechanisms to generate additional prompt variants for optimization similar to our approach, none have explicitly investigated code security. As a result, they lack key components such as security-focused scoring functions and mutation strategies tailored to secure code generation. To the best of our knowledge, our work is the first to introduce a GA-based prompt optimization framework that systematically integrates security-specific objectives.

2.3. Improving security of LLM-generated code

Recent studies have explored different techniques to tackle the shortcomings of LLMs in generating secure code. **Tuning-based.** He and Vechev (2023) presented a method based on prefix-tuning, where the LLM's weights remain unchanged while continuous task-specific vectors, known as prefixes, are learned to steer the model toward generating code with desired properties such as security. SafeCoder (He et al., 2024) combines instruction tuning with security-centric fine-tuning using datasets of secure and vulnerable code to encourage secure code generation. Similarly, CoSec (Li et al., 2024) introduces an on-the-fly security hardening technique that uses a separately fine-tuned small security model to guide a larger base model during generation.

Prompting Approaches. Tony et al. (2025a) explored different techniques that involved directly prompting an LLM and found that a technique called Recursive Criticism and Improvement (RCI) led to remarkable improvement in the security of LLM-generated code. Another similar study by Bruni et al. (2025) also explored prompt engineering techniques for secure code generation and arrived at the same conclusion that RCI offers the best performance for this usecase. **Retrieval-augmented Generation (RAG).** Additionally, Zhang et al. (2024) proposed SecCoder, which retrieves secure code examples from a database to guide generation. Although this method avoids retraining, its effectiveness depends on the diversity and coverage of the database across different programming tasks and languages. RAG was also employed by Tony et al. (2025b) where they retrieved secure coding guidelines applicable to a given coding task and used these guidelines to enhance the code generation prompt. This approach demonstrated comparable performance with RCI, the state-of-the-art prompting approach with reduced time and token consumption.

Discrete Prompt optimization. There is not a large collection of work that focus on optimizing discrete prompts, i.e., prompts written in human-readable tokens in natural language, for enhancing the security of LLM-generated code. Among the existing ones, the most related work for our paper is PromSec (Nazzal et al., 2024a), proposed by Nazzal et al., that introduces an approach that leverages a generative adversarial graph neural network to reduce security vulnerabilities in LLM-generated code. This refined code is then used to reverse-engineer security-aligned prompts, thereby optimizing discrete prompts given to the LLM for code generation. We provide a performance comparison of our approach with PromSec in Section 6.2. To the extent of our knowledge, there are no existing works that either explore the application of GA for secure code generation, or propose security-specific prompt mutations.

3. Methodology

In this study, we explore the application of GA to find optimal discrete prompts that can generate secure code. More specifically, our goal is to assess how the inclusion of security-specific prompt mutation techniques, in addition to generic ones, affects the performance of prompts optimized through a GA-based framework for secure code generation tasks. We are also interested in understanding the generalizability of the optimized prompts in this manner. Based on these goals, we formulated the following research questions.

RQ1: *Compared to generic mutation techniques, how does the addition of security-specific prompt mutation techniques impact the security of LLM-generated code?*

RQ2: *Do the prompts optimized using one dataset of coding tasks generalize to another dataset?*

RQ3: *How well do prompts optimized on one LLM transfer to other LLMs for secure code generation?*

The following subsections present the discrete prompt optimization pipeline employed in our study, followed by a detailed description of the prompt mutation techniques.

3.1. Prompt optimization for secure code generation

While prior works have used GAs for prompt optimization (Prasad et al., 2023; Xu et al., 2022), they primarily focus on natural language processing tasks such as language inference, question answering, and sentence completions. A direct application of these approaches to secure code generation use cases is not possible due to fundamental differences in task objectives, which necessitate customized scoring functions and other tailored components.

3.1.1. Problem formalization

An LLM query for code generation typically consists of 2 parts: an *instruction prompt* containing a phrase or sentence that guides the language model to perform the code generation and the *coding task* that specifies the functionalities to be implemented in the code. For example, ***"[prompt]*** *Generate Python code for the following task:* ***[task]*** *The code implements a user login page that takes user credentials as input."* Starting from a few instruction prompts that are used as seeds (and representing the 'standard' way a developer would communicate to the LLM), the purpose of our approach is to output new prompts that are optimized for secure code generation regardless of the coding tasks used. The optimization is performed on a limited number of reference coding tasks. The expectation is that, once the optimization process is completed, the optimized prompt instructions can be used for new and unseen coding tasks and they will still yeld the benefits of fewer weaknesses in the LLM-generated code.

This optimization problem is formalized as follows. Let $\mathcal{P}$ denote the space of candidate instruction prompts, and $\mathcal{X}$ the set of reference coding tasks. Then, we define a generation function $G(p, x)$ that, given the instruction prompt $p \in \mathcal{P}$ and coding task $x \in \mathcal{X}$, generates code $c \in C_p$, where C_p is the union set of all generated code samples for a given prompt p. Finally, we define the scoring function $S(C_p)$, or S_p in short, which quantifies the fitness of prompt p based on the code generated using it. With prompt optimization, we aim at finding the best instruction prompt $p^\star$ that produces the most secure code. Thus, we aim to solve the following optimization problem,

$$p^\star = \arg\min_{p \in P} S(C_p) \tag{1}$$

where C_p is defined as,

$$C_p = \bigcup_{x \in X} G(p, x) \tag{2}$$

Note that, to avoid evaluation of all possible prompts, the set of system prompts $\mathcal{P}$ can be limited into a finite predefined set, or be iteratively evolved as a population of a genetic algorithm.

3.1.2. Optimization pipeline

Based on the optimization problem defined in Eq. 1, we created a prompt optimization pipeline adapting GA for secure code generation, as shown in Fig. 1. This pipeline has 5 main components and their functionalities are explained below.

3.1.2.1. ***Query Preparation.*** The optimization process requires the availability of a small number of coding tasks that are used as reference for the optimization. These could come from the daily practice of developers belonging to an organization or software project. In this paper, we borrowed 36 coding tasks from an existing dataset (Tony et al., 2023). More details of this dataset and the tasks are provided in Section 4.1. In the first optimization step, our approach also requires a set of seed prompts. These could be manually provided by developers as well. In this work, we used a set of 5 seed prompts that we crafted according to the zero-shot prompt pattern as shown in Table 1. Our previous work (Tony et al., 2025a) explored different variations of zero-shot prompts for secure code generation. Among these, the *naive-secure* and *comprehensive* variants demonstrated the most potential, owing to their task-agnostic formulation and stronger performance in generating secure code. In the *naive-secure* variant, the term "secure" is added to the prompt to encourage secure implementations, whereas in the *comprehensive* variant, the prompt requests the LLM to prevent all the top security weaknesses listed in the CWE (Common Weakness Enumeration) published by MITRE (Mitre, 2024). We used these two variants to craft the 5 seed prompts used in our pipeline.

In the first iteration ($t = 0$), the query preparation component prepends each seed prompt to each coding task and passes the resulting queries to the code generation component. In the following iterations ($t > 0$), and up to T rounds, the query preparation components uses the prompts provided by the prompt mutation component.

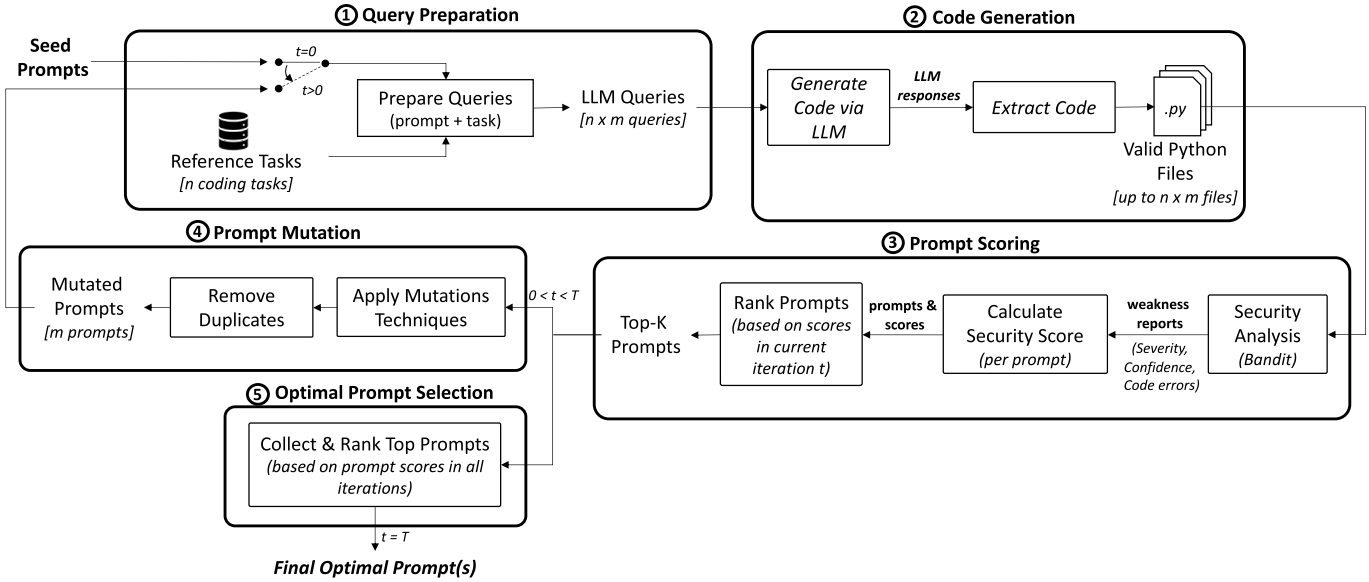

Fig. 1. GA-based prompt optimization pipeline tailored for secure code generation.

Table 1
Seed prompts used in the prompt optimization pipeline.

No.	Prompt
1	Generate secure Python code for the following task description:
2	Write Python code for the following task description that prevents security vulnerabilities:
3	Generate Python code that prevents all CWEs for the following task description:
4	Write Python code for the following task description that is very secure:
5	Generate a secure Python implementation that avoids the top security weaknesses in CWE for the following description:

3.1.2.2. ***Code Generation.*** The next step in the pipeline is to use an LLM to generate code for the functionalities specified by the queries. In this work, we experiment with GPT-3.5 and GPT-4, CodeLlama, Gemini, and DeepSeek-Coder. Moreover, we focus on code generated in Python due to the widespread use of this programming language (TIOBE Index, 2024; PYPL Index, 2024; GitHubOctoverse, 2024) as well as its continued popularity in security research (Rauf et al., 2022; Rahman et al., 2019; van Oort et al., 2022). The responses generated by the LLM models are further processed to isolate the code segments from any spurious text generated by the LLM. This step is necessary as the model-generated responses frequently included additional textual content, such as extensive code explanations.

3.1.2.3. ***Prompt Scoring.*** Once code is generated for the reference tasks, the prompts used for this generation must be scored based on the optimization problem defined in Eq. 1. As our use case is secure code generation, we use code security as the criteria to score the prompts. For this, we employed PyCQA (2025), a static analysis tool specifically engineered to detect security weaknesses in Python code, to assess the security of the LLM-generated code. Bandit was chosen due to its use in several prior studies (Rahman et al., 2019; Rauf et al., 2022; Ruohonen et al., 2021; Nazzal et al., 2024b) for detecting security weaknesses in Python. Bandit examines the code by building abstract syntax trees (ASTs) from it and provides a report detailing the number of weaknesses, their descriptions, associated CWE IDs, severity, and confidence levels. Severity levels are determined by the impact and likelihood of the weakness, while confidence levels are based on how certain Bandit is about the presence of the weakness in the code. Bandit assigns severity and confidence levels-*low, medium*, or *high*-for each detected security weakness. It also reports issues in code in the form of syntactical and structural errors, incorrect construct usage, and undefined variables or functions that are encountered during the AST parsing. To evaluate the quality of each prompt in generating secure and correct code, we define a scoring function $S(C_p)$, or S_p in short, as shown below.

$$S_p = \sum_{c=1}^{n} \left(\sum_{j=0}^{w} \left(\text{Severity}_{(c,j)} \cdot \text{Confidence}_{(c,j)} \right) + \delta_c \right) + \delta_{aggregate} \tag{3}$$

Here, C_p is the set of code generated by prompt p for the reference tasks (see Eq. 2). S_p is computed as an aggregate security risk score over all n code samples in C_p based on the security weaknesses and errors contained in them. For each code sample $c \in C_p$, the security risk is calculated as the sum of the product of the severity and confidence levels ($low=1$, $medium=2$, $high=3$) reported by Bandit for each of the w reported weaknesses. Additionally, the score also adds a penalty δ_c which accounts for the presence of errors or other issues encountered during the static analysis of code sample c. This is to ensure that the prompt yields valid, compilable code instead of incoherent and malformed output. It was observed that some of these errors can occur in code independently of the prompt used and are attributable instead to the token continuation generated by the LLM during that specific request. However, if such errors are prevalent across a substantial number of code responses generated from the same prompt, this suggests that the prompt itself may be a contributing factor to the observed issues. Hence, in addition to δ_c, the prompts that led to errors in more than half of n generated code samples received an additional penalty $\delta_{aggregate}$ to further penalize prompts that generally led to error-prone code responses. A lower score indicates better prompt performance, as it reflects fewer and/or less severe weaknesses, higher syntactic correctness, and overall better adherence to secure coding practices. Once the score of the prompts in iteration t is calculated, the prompt scoring component selects the top K prompts with the lowest S_p values and passes them on to the next step, following the truncation selection criteria (Onakpojeruo et al., 2024) widely adopted in GAs. The K prompts along with their scores are also passed to the optimal prompt selection component (see later).

3.1.2.4. ***Prompt Mutation.*** The prompt mutation stage constitutes a crucial component of the genetic algorithm. The prompt mutation module receives K input prompts and generates a number of mutated prompts as output. In this paper, we employ two types of mutation techniques: *generic* and *security-specific*, each described in detail in Section 3.2. Following the mutation, all duplicate prompts generated up to the current iteration are removed, ensuring a unique population. This newly generated set of m mutated prompts is then passed to the query preparation stage, and the optimization cycle is repeated for T iterations.

3.1.2.5. ***Optimal Prompts Selection.*** After T iterations, the optimization process terminates, and the best-performing prompts, i.e., those with the lowest scores across the top-K prompts from all iterations, are returned as the final set of optimized prompts. These prompts can be used by developers as is for future coding tasks, as we show that they generalize well in this respect.

3.2. Prompt mutation techniques

Most existing prompt optimization approaches (Prasad et al., 2023; Lester et al., 2021; Xu et al., 2022) rely on general-purpose or generic prompt mutation strategies that are agnostic to the specific task or domain. In this work, we investigate whether such generic techniques are sufficient to enhance the security of code generated by LLMs or if incorporating domain-specific, security-oriented mutations can further improve performance. The mutation techniques used in both categories are described in detail below.

3.2.1. Generic techniques

Existing GA-based prompt optimization approaches utilize a range of mutation techniques, including back translation (Xu et al., 2022; Longpre et al., 2020), cloze (Xu et al., 2022), sentence continuation (Xu et al., 2022), paraphrasing (Prasad et al., 2023; Hao et al., 2023; Jiang et al., 2020), model-driven mutation and crossover (Guo et al., 2024), as well as edit-based operations such as addition, deletion, and swapping of words or phrases (Prasad et al., 2023). In our preliminary analysis, we observed that the edit-based techniques performed poorly for our use case involving short prompts, often resulting in semantically incoherent or irrelevant variants that led to a large number of invalid code generation. Consequently, these methods were excluded from further evaluation. Additionally, we observed that paraphrasing, sentence continuation, mutation, and crossover largely performed the same operation of modifying a prompt's surface form without altering its core semantics, producing overlapping results when applied via language models. Based on these insights, we selected three distinct and representative general-purpose mutation techniques for evaluation in our pipeline: back translation, paraphrasing, and cloze.

Back Translation: This is a commonly used approach for data augmentation (Xu et al., 2022; Longpre et al., 2020), where an English phrase is translated into other languages and then translated back into English to produce different variations of the original phrase. In our experiments, we translated the prompts into French, German, Spanish, and Italian and then back to English. We used Opus MT pre-trained models (Opus-MT, 2024), which use the Marian NMT framework for generating translations.

Paraphrase: In this approach, a model is tasked with paraphrasing a given prompt without altering its meaning to generate more variations. This is also a popular approach used for data augmentation (Prasad et al., 2023). Paraphrasing was done using the T5 model (T5 Paraphrase Paws, 2024), `Vamsi/T5_Paraphrase_Paws`, specialized for generating paraphrases of English sentences. In this study, we generated four paraphrases of the given prompt.

Cloze: Here, random tokens within the prompts are replaced by placeholders (Xu et al., 2022). A model is then used to fill in these placeholders based on the context provided by the rest of the prompt. Given the generally short length of the prompts in our experiment, we replaced a maximum of two randomly chosen tokens. We repeated this process 4 times per prompt to get 4 different cloze variations of each prompt. This approach was carried out using `bert-base-uncased`, which is a BERT model (BERT, 2024) since it is a masked LLM.

3.2.2. Security-specific techniques

To facilitate prompt variants that explicitly incorporate security-relevant information, we introduce two security-specific, LLM-assisted prompt mutation techniques: *self-guided* and *feedback-guided* mutation. The goal is to generate more secure versions of a given code generation prompt. The most direct approach to achieve this is by leveraging the capabilities of an LLM itself. In particular, in this paper, we use `gemini-pro` as the prompt mutation LLM.

Self-guided Secure Prompt Mutation: In this technique, an LLM is prompted to transform an existing code generation prompt into a more secure version, one that is likely to produce code free of security vulnerabilities. After initial experimentation with several versions, Fig. 2 (left) shows the resulting instruction supplied to the LLM to generate a prompt mutation. The three notes appended to the instruction prompt were informed by preliminary experimentation with various prompt formulations. Initially, the LLMs frequently generated code snippets instead of rewriting the prompt itself, which motivated the inclusion of the first note. Additionally, in some instances, the models introduced their own coding tasks and then generated prompts specific to those tasks; to address this, the second note was added. Lastly, the third note was included to constrain the verbosity of the generated output and prevent the production of excessively long prompts. An example of a prompt generated using the self-guided techniques is given below:

Self-guided Mutation

Generate a security-enhanced variation of the following *CODE_GENERATION_PROMPT* to prevent security vulnerabilities in the generated code.

CODE_GENERATION_PROMPT: <*prompt to be modified*>

Note 1: Do not write any code, only write the modified prompt.
Note 2: The prompt should be task-agnostic, i.e, applicable to any coding task.
Note 3: The modified prompt should be concise and under 250 words.

Feedback-guided Mutation

Generate a security-enhanced variation of the following *CODE_GENERATION_PROMPT* by incorporating the following *SECURITY_WARNINGS* generated by a SAST tool while evaluating the code generated for different coding tasks.

SECURITY_WARNINGS: <*selected warnings*>
CODE_GENERATION_PROMPT: <*prompt to be modified*>

Note 1: Do not write any code, only write the modified prompt.
Note 2: The prompt should be task-agnostic, i.e, applicable to any coding task.
Note 3: The modified prompt should be concise and under 250 words

Fig. 2. Prompt templates used for the generation of security-enhanced code generation prompts using the *self-guided (left)* and *feedback-guided (right)* techniques.

> **Original Prompt**: Generate secure Python code for the following task description.
>
> **Mutated Prompt**: Generate a secure Python implementation for the following task description, ensuring that the code is free from common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and buffer overflows. Additionally, the code should adhere to best practices for input validation, error handling, and data sanitization.

Feedback-guided Secure Prompt Mutation: In this technique, rather than relying solely on an LLM to integrate security considerations into the prompts, we leverage the feedback from a SAST tool to guide the process. Specifically, we aggregate all the security warnings generated by Bandit for all the reference tasks during each iteration in the optimization pipeline and identify the top five most frequently occurring warnings. These warnings, along with the code generation prompt to be modified, is provided to the mutation LLM, which is then instructed to rewrite the prompt into a more secure version by incorporating the given security insights. The template used for the prompt mutation is shown in Fig. 2 (right). An example of a prompt generated using the feedback-based technique is shown below:

> **Original Prompt**: Write Python code for the following task description that is very secure.
>
> **Mutated Prompt**: Write very secure Python code for the following task description, taking into account the following security warnings: Avoid binding to all interfaces, consider the security implications of using the subprocess module, set a timeout when using the requests module, use a strong hash function (e.g., SHA-256) instead of MD5 for security purposes, avoid using a shell when starting a process to prevent injection attacks.

4. Experimental design

Here, we provide details of the datasets used for optimization and evaluation, as well as the models employed for code generation. We then outline the hyperparameters configured for the optimization pipeline, followed by a description of the evaluations and analyses conducted.

4.1. Datasets of coding tasks

The optimization pipeline in Section 3.1 requires a reference set of secure coding tasks. For this, we used a subset of coding tasks from a dataset called **LLMSecEval** (Tony et al., 2023), designed specifically for assessing the security of code generated by LLMs. LLMSecEval consists of 150 coding tasks in the form of natural language descriptions, covering 18 of the Top 25 CWEs.

A key advantage of using GAs for prompt optimization is their effectiveness even with a small number of reference examples (Prasad et al., 2023; Xu et al., 2022; Zhao et al., 2023; Guo et al., 2024), in contrast to tuning- or learning-based approaches that typically require large datasets (Lester et al., 2021; Wang et al., 2023). For our experiments, we use 36 coding tasks (2 tasks for each CWE type) as the reference set. The remaining 114 tasks in LLMSecEval are used for the evaluation of the optimized prompts (RQ1). Additionally, we use a second dataset, **SecurityEval** (Siddiq and Santos, 2022), to evaluate the prompts optimized on LLMSecEval on an unseen dataset to answer RQ2. SecurityEval contains 121 coding tasks in the form of incomplete code snippets paired with docstring comments that specify the code functionality. The dataset covers 69 CWEs, with each task designed to target a specific type of weakness. Examples of coding tasks from both datasets are provided in A.

4.2. Code generation models

As mentioned in Section 3.1.2.2, we used five LLMs in our experiments. Since, the OpenAI models are widely adopted in the majority of existing studies on code generation (Rabbi et al., 2024; Luo et al., 2025), including several works that explicitly explore prompt optimization (Nazzal et al., 2024b; Cheng et al., 2024; Tony et al., 2025a), we included GPT-3.5 and GPT-4 in our evaluations. For GPT-3.5 and GPT-4, we used the *gpt-3.5-turbo* model (released in March 2023) and the *gpt-4-1106-preview* model (released in November 2023), both accessed through the official API. In order to not restrict our evaluations to just OpenAI models, we also chose the Gemini model from Google that also has demonstrated good coding capabilities (Luo et al., 2025; Wang et al., 2025; Zheng et al., 2024a). For Gemini, we used `gemini-1.5-flash` (released in May 2024) via its API. The APIs for GPT-3.5 and GPT-4 were accessed between March and May 2024, whereas Gemini was accessed in November 2024. Additionally, to ensure balanced representation between proprietary and open-source models, we included CodeLlama from Meta, using the *codeLlama-7b-hf* version (released in July 2023, 7B parameters), a smaller but code-specialized open-source model known for its efficiency on limited resources (Wang et al., 2025; Zheng et al., 2024a). We further incorporated DeepSeek-Coder with the `deepseek-coder-33b-instruct` version (released in 2024, 33B parameters), an open-source model that has attracted significant attention for its strong coding capabilities (Luo et al., 2025; Wang et al., 2025; Zheng et al., 2024a). To ensure maximum reproducibility, we set the temperature to 0.0 (Tony et al., 2025a) and the top_p to 0.1 (Tony et al., 2025a) across all models.

4.3. Evaluation of optimized prompts

We evaluated two variants of our GA-based prompt optimization: (i) *GA generic*, which applies only generic mutation techniques, and (ii) *GA complete*, which integrates both generic and security-specific mutation techniques. We also used the manually written seed prompts as a *baseline* to compare the performance of the optimized prompts. After completing the two types of prompt optimization for each of the five LLMs, we selected the top five optimal prompts per model from both optimizations. These prompts, tailored to their respective LLMs, were then used to generate code for 114 tasks from the LLMSecEval (**RQ1**) and 121 tasks from the SecurityEval (**RQ2**). The generated code first underwent a preliminary check for compilability using Python's `py_compile` library to ensure that the responses generated by the LLMs are compilable code without any syntax or other errors. The compilable code samples were subsequently analyzed for security weaknesses. For the security analysis, in addition to Bandit, we also used a tool called CodeQL. CodeQL analyzes vulnerabilities by converting source code into a database and applying a declarative query language to identify issues and is widely used in existing research (Pearce et al., 2022, 2023; Siddiq et al., 2024). CodeQL was added to improve the reliability of our findings, especially because we used Bandit in our optimization pipeline.

4.4. Transferability to other LLMs

We conducted additional evaluations to understand if the prompts optimized on one model are transferable to other models (**RQ3**). For this, the top 5 optimal prompts from each LLM were used to generate code using the other 4 LLMs. We used the 114 tasks from LLMSecEval for this additional evaluation, followed by a compilability check and the subsequent analysis of the generated results for security weaknesses by Bandit and CodeQL.

4.5. Manual validation for functional correctness

Since LLMSecEval and SecurityEval do not provide test cases to assess functional correctness, we manually verified this aspect for a sam-

Table 2
Summary of the hyperparameter values and other details of the GA-based optimization pipeline.

Population size (initial)	5 seed prompts
Number of reference tasks	36
Selection strategy	Truncation (top-K)
K	4
Number of iterations	6
δ_c	10
$\delta_{\text{aggregate}}$	100
Number of mutation techniques	
GA generic	3
GA complete	5
Number of Mutations per technique	4

ple of all LLM-generated code samples. The evaluation for code validity followed the methodology proposed by Tony et al. (2025a), which defines functional correctness based on two key criteria: *task alignment* that checks whether the code fulfills the functional requirements outlined in the task description and *completeness* that checks whether the code includes complete implementation of the functionality specified in the task instead of just placeholders like empty functions or natural language comments. The detailed verification procedure is described in Tony et al. (2025a). Since performing this manual code validity check for the entire results was not feasible due to the large number of generated code samples (38,005 samples), we evaluated 40 % of all the code samples and report the results.

4.6. Manual validation of security analysis results

Bandit is a tool that applies rule-based checking on ASTs of code to detect different vulnerabilities in Python. Although a comprehensive list of weaknesses covered by Bandit is unavailable in its documentation, the main areas covered include, but not restricted to, injection flaws, cryptographic weaknesses, hard-coded credentials, insecure permissions and use of insecure libraries. CodeQL on the other hand, models code as relational databases and runs custom or prebuilt queries to identify weaknesses. CodeQL checked for around 30 weakness categories mainly including injection flaws, path traversal, sensitive data exposure, insecure deserialization and insecure hashing algorithms.

Since the security evaluation of LLM-generated code relies on Bandit and CodeQL, we chose to manually verify the reliability of these tools for this purpose. This verification was conducted on the results produced by all five LLMs across all prompting approaches. Due to the manual nature of this assessment, we limited the evaluation to a sample of the results, covering a total of 300 code samples. For each code file, we manually reviewed every security warning reported by Bandit and CodeQL, checking the corresponding code to determine whether the warning was a false positive. If a vulnerability was confirmed to be present, it was documented in our manual inspection results; otherwise, it was considered a false positive and excluded. Weakness description from the SAST results, the associated CWE ID and additional information regarding the identified CWE from MITRE documentation were used to determine the presence of the weakness.

4.7. Hyperparameters for optimization

Table 2 presents the hyperparameters and configuration details used to execute the optimization pipeline. The initial GA population size (5 prompts), the number of reference tasks (36 tasks), and the candidate selection strategy (truncation strategy) have already been described in earlier sections. Truncation strategy uses either a fixed percentage or an absolute number of candidates for the next generation. We used an absolute number K instead of a percentage in order to control the number of mutations, mainly due to resource restrictions. Thus, we set $K = 4$ so that most high-performing prompts were retained from the seeds, while still allowing turnover through mutation to introduce variation across generations. In a preliminary assessment that studied the optimization behavior over 20 iterations, we observed that the prompt optimization pipeline gave the best results in terms of code quality when the number of iterations T was set to 6. Hence, we set $T = 6$ for all our experiments. The details of this preliminary assessment are provided in B. The penalty δ_c added to the prompt score S_p for generating code sample c containing errors (see Section 3.1.2.3) was set to 10, while the aggregate penalty $\delta_{aggregate}$ when the prompt resulted in erroneous code for more than half of the generated samples was set to 100. The $\delta_{aggregate}$ was set significantly higher than δ_c to ensure that prompts generating a large number of error-prone code responses were less likely to be selected as top prompts from all iterations. The *GA generic* optimization employs three mutation techniques: back translation, paraphrasing, and cloze transformation, while the *GA complete optimization* incorporates two additional security-specific techniques: self-guided and feedback-guided mutation. Each technique generates 4 mutated prompts, resulting in up to 48 mutated prompts per iteration for *GA generic* and up to 80 for *GA complete optimization*, depending on the removal of duplicates.

5. Results

We generated code using 5 LLMs, which are prompted using 3 different approaches, which are *baseline*, *GA generic* and *GA complete*. In the first approach (baseline), we use the 5 manually written prompts in Table 1, which are also used as seeds in Figure1. This case represents the baseline, i.e., when our optimization approach is not used. In the second approach (GA generic), we execute the optimization pipeline, but only using the generic mutation techniques, i.e., the security mutation techniques are turned off, as mentioned in Section 4.3. Accordingly, we use the top 5 optimal prompts that are produced. In the third approach (GA complete), we use the top 5 optimal prompts that are produced by the optimization approach, when both generic and security-specific techniques are used for prompt mutation.

5.1. Security impact of generic and security-specific mutation techniques

The code samples generated using *baseline* prompts, as well as prompts optimized through both optimization settings (GA generic and GA complete), were analyzed for security weaknesses using Bandit and CodeQL. Table 3 presents the results of this analysis for the code generated for 114 hold-out coding tasks from the LLMSecEval dataset by the five LLMs evaluated in this study. The leftmost column of the table indicates the approach used for code generation, where we used 5 prompts from each approach for code generation, resulting in a maximum of 570 code samples per approach. However, only valid code samples produced by the LLMs were included in the security analysis. Therefore, the second column (no.valid code) reports the number of syntactically and structurally correct (valid) code snippets produced in each setting. As discussed in Section 4.5, we manually validated the functional correctness of 40 % of all the generated code. Among the code that was syntactically and structurally valid, we found that 2.86 % was functionally incorrect. This margin of error should be taken into account when interpreting the results. More details on the functional correctness per LLM is presented in the Appendix C. The third column (average LOC) report the average lines of code (LOC) of the valid samples. Security weaknesses identified by Bandit and CodeQL are reported as: (i) the number of code samples containing at least one weakness (no.vuln code), (ii) the average number of weaknesses per code sample (rate), and (iii) the average number of weaknesses per generated LOC (density). Additionally, Table 4 shows the frequency of specific types of weaknesses (CWE) detected by both Bandit and CodeQL in the LLM-generated code. It should be noted that we excluded weaknesses that occurred fewer than two times across all settings from the table to maintain readability and avoid sparsely populated rows. The full set of results is available in the replication package.

Table 3
Security analysis of code generated by 5 LLMs, using the three approaches for the hold-out tasks in the LLMSecEval dataset. Rate = average number of weaknesses per code sample, density = average number of weaknesses per generated line of code.

LLMSecEval dataset (hold-out tasks)								
GPT-3.5								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	551	17.30	159	0.471	0.035	98	0.245	0.012
GA generic	459	12.58	86	0.281	0.023	37	0.095	0.006
GA complete	569	21.05	96	0.244	0.013	106	0.223	0.009
GPT-4								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	568	23.51	231	0.718	0.032	177	0.457	0.016
GA generic	563	24.44	187	0.522	0.022	152	0.422	0.013
GA complete	567	35.82	79	0.319	0.01	113	0.276	0.008
CodeLlama								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	500	14.32	180	0.626	0.058	100	0.246	0.016
GA generic	491	11.83	163	0.556	0.068	80	0.183	0.018
GA complete	504	38.67	166	0.463	0.02	59	0.154	0.005
Gemini								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	570	31.77	212	0.598	0.018	169	0.470	0.012
GA generic	570	32.33	158	0.480	0.018	140	0.342	0.008
GA complete	567	43.01	134	0.363	0.01	132	0.294	0.006
DeepSeek-Coder								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	569	16.79	316	0.871	0.057	249	0.528	0.028
GA generic	566	13.87	283	0.740	0.071	220	0.441	0.028
GA complete	570	24.79	124	0.349	0.016	73	0.161	0.008

Bandit and CodeQL differ substantially in the types of security weaknesses they detect, primarily due to differences in their underlying analysis techniques and rule sets. For instance, Bandit identified instances of CWE-20 in scenarios involving the parsing of untrusted XML input without appropriate validation. In contrast, CodeQL reported CWE-20 in cases where the code lacked proper HTML filtering or failed to sanitize URLs effectively. Given these distinct detection scopes, we analyze the results from each tool separately.

5.1.1. Bandit results

Based on the results of the security analysis with Bandit, both *GA generic* and *GA complete* prompt sets consistently resulted in a lower weakness rate in code generated by all models compared to the *baseline* prompts. The generic prompts, optimized using simple mutation techniques such as back translation, cloze, and paraphrasing, led to a notable reduction in weakness rate, especially for GPT-3.5 (-40.33 %) and GPT-4 (-27.29 %), highlighting the effectiveness of even basic prompt mutation strategies. Specifically, notable reductions can be observed in CWE-94: *Code Injection*, CWE-78: *Command Injection*, and CWE-259: *Use of Hard-coded Passwords* for code generated by GPT-3.5 and GPT-4, as shown in Table 4. Instances of CWE-94 are frequently recorded when a Flask application is run in debug mode, which includes a feature permitting arbitrary code execution. The models removed this weakness by running the Flask application using `app.run(debug=False)` in many instances. In the LLM-generated code, CWE-78 predominantly occurs when an operating system command is initiated with a partial executable path or when a `subprocess.run()` command is invoked using user-provided input. The models reduced this weakness by using built-in functions such as `os.listdir()` function instead of executing the *ls* command with a user-provided path to list files. CWE-259 is often attributed to the usage of hard-coded credentials during database connection and application login checks, along with the utilization of hard-coded secret keys for session management. In many cases, these were avoided by GPT-3.5 and GPT-4 by obtaining credentials from the environment variables. Aside from these cases, different LLMs showed varying behavior across specific weakness types when evaluated with the prompts from the three settings. The least improvement was exhibited by CodeLlama where the weakness rate reduced only by 11.18 %.

Like the *GA generic* prompts, the *GA complete* prompts notably reduced weaknesses across all LLMs except CodeLlama. Furthermore, incorporating security-specific mutation techniques led to further improvements, resulting in additional reductions in weakness rates across all LLMs when compared to the *GA generic* prompt set. The most substantial improvements were observed in code generated by GPT-4 and DeepSeek-Coder, with reductions of 38.88 % and 52.83 %, respectively. Similar to the *GA generic* prompt set, these reductions are largely driven by a decrease in the occurrence of CWE-94 and CWE-259. Additionally, notable improvements were observed in mitigating CWE-330: *Use of Insufficiently Random Values*, which was commonly triggered by the use of standard pseudo-random number generators not suitable for crypto-

Table 4
Number of specific weaknesses detected by Bandit and CodeQL in code generated by the LLMs for the LLMSecEval dataset (hold-out tasks).

LLMSecEval dataset (hold-out tasks)															
Bandit															
	GPT-3.5			***GPT-4***			***CodeLlama***			***Gemini***			***DeepSeek-Coder***		
CWE	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete
20	0	0	0	0	0	0	4	7	6	0	0	0	0	0	0
22	13	4	8	2	0	5	9	7	2	9	4	3	10	10	6
78	77	50	50	112	84	105	85	84	43	79	93	71	100	84	78
89	2	0	1	0	0	0	17	6	15	0	0	1	5	9	3
94	16	5	0	146	87	0	47	34	11	110	39	2	232	188	1
259	60	47	38	88	75	33	65	53	91	49	40	38	68	47	30
327	20	1	3	0	0	1	6	2	5	0	0	0	3	5	1
330	59	11	27	39	33	2	57	67	39	18	56	21	64	65	52
377	10	9	8	15	10	12	12	14	9	42	28	29	12	10	16
400	4	6	0	3	4	1	10	4	7	3	1	0	1	1	0
502	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
605	0	0	0	2	1	9	1	1	4	31	12	41	0	0	12
703	0	0	4	2	0	13	0	0	3	0	1	0	0	0	0
CodeQL															
	GPT-3.5			***GPT-4***			***CodeLlama***			***Gemini***			***DeepSeek-Coder***		
CWE	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete
20	5	2	5	4	5	5	1	3	5	0	0	0	5	9	8
22	15	12	12	5	10	10	19	7	2	0	17	16	3	7	21
78	0	1	7	1	2	2	2	1	1	14	0	0	0	1	5
79	6	2	6	4	4	5	5	4	7	4	6	4	4	7	14
89	2	0	0	0	0	0	2	0	0	0	0	0	6	5	3
200	42	13	10	172	130	50	53	40	33	196	98	30	242	195	17
327	51	17	76	5	9	6	40	31	28	38	64	91	12	15	10
400	0	0	0	4	6	9	0	0	0	1	1	2	0	0	0
502	0	0	0	7	8	5	5	4	1	0	0	0	1	1	0
601	14	6	11	59	63	64	2	3	1	15	9	24	27	9	14

graphic applications. This weakness was effectively addressed by leveraging the `secrets` library for secure random number generation. While GPT-3.5 showed substantial reductions in weaknesses compared to the *baseline* prompt set, the gains over the *GA generic* set were relatively modest (-13.16 %), the lowest among all models. Table 3 also shows an increase in the number of code samples with weaknesses, from 86 to 96, when using the *GA complete* prompts compared to the *GA generic* prompts. However, this increase is primarily due to the larger number of valid code samples generated with the *GA complete* prompt set. CodeLlama exhibited the second-lowest improvement (-16.72 %), which may be attributed to the use of its smaller 7B variant in this study, potentially limiting the effectiveness of prompts optimized through both mutation techniques.

5.1.2. CodeQL results

Similar to the results from Bandit, the CodeQL results in Table 3 shows that the *GA generic* prompt set consistently reduced the weakness rate across all LLMs compared to the *baseline*. We saw further reductions in weakness rates for the *GA complete* prompts, though GPT-3.5 remained an exception to this pattern. For GPT-3.5, the *GA generic* prompts resulted in a significant reduction in the weakness rate (0.095), outperforming both the *baseline* (0.245) and *GA complete* (0.223) prompt sets. This improvement is primarily driven by notable decreases in occurrences of CWE-327: *Use of a Broken or Risky Cryptographic Algorithm* and CWE-601: *URL Redirection to Untrusted Site*, as can be seen in Table 4. CWE-327 was reported when a weak hashing algorithm like SHA256 and MD5 was used in code, while CWE-601 was reported when a URL used for redirect was constructed using user-provided data. Examining the CWEs reported by Bandit, we find that CWE-327 and CWE-601, two of the most prominent weaknesses identified by CodeQL, appear infrequently or not at all. Since the prompt optimization pipeline relied on Bandit's analysis, this may explain the inconsistent behavior of these specific weakness types. It should also be noted that this pattern is not observed in the outputs of the other LLMs. Another prominent weakness detected by CodeQL is CWE-200: *Exposure of Sensitive Information to an Unauthorized Actor*. The *GA complete* prompts consistently reduced this weakness compared to both *baseline* and *GA generic* prompt sets. Similar to the Bandit results, most notable reduction in weakness rate due to the *GA complete* prompts is observed in *GPT-4* (-34.59 %) and DeepSeek-Coder (-63.49 %) compared to the *GA generic* prompts.

5.1.3. Statistical tests

We also performed a Kruskal-Wallis test (Kruskal and Wallis, 1952) followed by Post-Hoc Dunn's test (Dunn, 1961) with Bonferroni correction (corrected significant level (α) = 0.05/3 = 0.01667) on the results for each LLM to determine the statistical significance of the results obtained for each prompt set. In the Bandit results, we saw that the *GA generic* prompt set significantly ($p < \alpha$) reduced the weaknesses in code generated by all the models except CodeLlama ($p = 0.313$) and DeepSeek-Coder ($p = 0.028$) when compared to the *baseline* prompts. On the other hand, the *GA complete* prompts attained significant reductions compared to the *baseline* prompts in all LLMs except CodeLlama ($p = 0.089$). The notable improvements seen in GPT-4 ($p = 2.72E-10$) and DeepSeek-Coder ($p = 0.0$) compared to the *GA generic* prompts are also substantiated by the statistical test results.

In the case of CodeQL results, we observed a statistically significant reduction in the weakness rate by the *GA generic* prompts only in code generated by GPT-3.5. As for the *GA complete* prompts, significant reductions were observed in code from all LLMs except GPT-3.5, when compared to the *baseline* prompts, showcasing the advantage of the *GA complete* prompts over the *GA generic* prompts. Furthermore, we

also observed significant reductions in weakness rate due to the *GA complete* prompts in code generated by *GPT-4* ($p = 4.53E-03$) and DeepSeek-Coder ($p = 0.0$) compared to the *GA generic* prompts.

RQ1: In our experiments, the use of a GA-based approach that combines both generic and security-specific mutation techniques always outperforms the direct use of the seed manual prompts. Furthermore, incorporating security-specific prompt mutation techniques into the optimization pipeline notably reduced security weaknesses in LLM-generated code compared to using only generic techniques. The impact was especially pronounced in the outputs of GPT-4 and DeepSeek-Coder.

5.2. Generalizability to different dataset

As the prompts are optimized on a subset of the coding tasks in the LLMSecEval dataset, we verified whether those same prompts are generalizable to other datasets by performing additional evaluations on the SecurityEval dataset. In the latter dataset, the tasks are specified in a different way, which might add an additional challenge. While LLMSecEval contains task descriptions in natural language only, SecurityEval contains tasks in the form of incomplete code snippets and code comments. Please, refer to A for examples of both task types. The results of this analysis are shown in Tables 5 and 6. Note that the SecurityEval dataset consists of 121 tasks. Hence, each prompt set generated up to 605 code samples.

5.2.1. Bandit results

The performance of the *GA generic* prompt set did not match its effectiveness observed in the LLMSecEval dataset. For GPT-3.5 and CodeLlama, these prompts resulted in a higher weakness rate than the *baseline* prompts. Among the remaining three LLMs, the *GA generic* prompts performed better than the *baseline* ones, though the improvement was minimal for DeepSeek-Coder, with only a 0.55 % reduction in weakness rate showing no significant improvement. In contrast, the *GA complete* prompts consistently outperformed both the *baseline* and *GA generic* sets in terms of weakness rates. Additionally, improvements were observed with DeepSeek-Coder, where the weakness rates significantly dropped by 43.53 %, compared to the *GA generic* prompts. The *GA generic* prompts are optimized using basic mutation techniques such as back translation, paraphrasing, and cloze. These methods, particularly back translation and paraphrasing, perform modifications primarily based on the existing prompt context, offering limited meaningful enhancements. In contrast, the *GA complete* prompts leverage techniques that incorporate SAST feedback and instruct the model to enrich the prompt with security-specific guidance. This results in more domain-specific improvements that generalize better across a wider range of security-related tasks, regardless of the specific coding task functionalities seen during training. Consequently, the *GA complete* prompts demonstrate better generalizability to new datasets compared to the *GA generic* prompts.

Table 6 shows the specific weakness categories detected by Bandit. Except for Gemini, the *GA complete* prompts reduced the occurrence of CWE-78 in the code generated by all the LLMs compared to the other two prompt sets following the same pattern observed in Section 5.1.1. Similarly, substantial reduction in CWE-94 can be seen in code generated by GPT-4, Gemini, and DeepSeek-Coder. This can be attributed to the fact that these two weaknesses are among the most frequently detected in code generated for tasks in the LLMSecEval dataset as well. Consequently, the prompts optimized using the GA-complete setting incorporate specific security cues aimed at mitigating these weaknesses, introduced through the feedback-guided mutation technique which leverages security warnings from Bandit generated on the reference tasks from LLMSecEval. We observed an increase in the frequency of CWE-20, in all the models when *GA complete* is used compared to *GA generic*. This weakness manifested in the code in the form of parsing of untrusted XML using insecure libraries such as `lxml` and `xml.etree.ElementTree` without proper validation. This may be because the tasks that involves parsing XML inputs are absent in LLMSecEval, and hence, none of the optimized prompts include any security cues to address this weakness. Regardless, the *GA complete* prompts have managed to reduce the overall frequency of security weakness, which shows promise, considering that SecurityEval covers much more weakness categories (69 CWEs) compared to that of LLMSecEval (18 CWEs).

5.2.2. CodeQL results

In the CodeQL results, the *GA generic* prompts failed to reduce the weakness rate in code generated by DeepSeek-Coder, while it performed better than the *baseline* prompts for the other LLMs, although not by a large margin. The highest reduction in weakness rate was observed in code generated by GPT-4 (-27.8 %). Substantiating the results from Bandit, the *GA complete* prompts performed better than *baseline* and *GA generic* prompts. The most notable reductions compared to the *GA generic* prompts are observed in the code generated by DeepSeek-Coder (-39.17 %). As can be seen in Table 6, when it comes to specific weakness categories, different LLMs behave differently to the three prompt sets. The *GA complete* prompt set notably reduced the frequency of CWE-200 in all the models except CodeLlama. However, both the *GA generic* and *GA complete* prompts led to an increased occurrence of CWE-327, stemming from the use of weak hashing algorithms, in the code generated by all LLMs except DeepSeek-Coder. As discussed in Section 5.1.1, this weakness was rarely flagged by Bandit during prompt optimization on the LLMSecEval subtasks, which likely resulted in insufficient tuning of the prompts to mitigate this issue.

5.2.3. Statistical tests

In the bandit results of the *GA generic* prompts, significant reductions were observed in GPT-4 ($p = 2.07E-08$) and Gemini ($p = 4.00E-05$) compared to the *baseline* prompts, whereas the *GA complete* prompts attained significant improvements for DeepSeek-Coder ($p = 8.73E-04$) as well along with GPT-4 ($p = 1.33E-15$) and Gemini ($p = 7.62E-10$). The *GA complete* prompts also saw significant improvements compared to the *GA generic* prompts in code generated by DeepSeek-Coder ($p = 0.00$)

In the results of the CodeQL analysis, the *GA generic* prompts managed to significantly reduce the weaknesses in code generated by GPT-4 ($p = 9.03E-04$) compared to the *baseline* prompts, while the *GA complete* prompts saw significant reductions in code from all the examined LLMs, except Gemini ($p = 0.037$). Furthermore, the *GA complete* prompts also showed significant improvements compared to the *GA generic* prompts in the code generated by DeepSeek-Coder ($p = 1.21E-04$).

RQ2: In the case of generic mutation techniques, prompts optimized on tasks from the LLMSecEval dataset (which are in the form of natural language descriptions) provided only limited improvement over the *baseline* prompts when applied to SecurityEval tasks (which are in the form of incomplete code snippets). In contrast, prompts optimized with security-specific mutation techniques demonstrated better generalizability, especially for DeepSeek-Coder, notably reducing weakness rates and thereby confirming their effectiveness on new datasets with unseen types of coding tasks and weaknesses.

5.3. Transferability to other LLMs

To evaluate how prompts optimized on one model apply to others, we used the prompts optimized on each model (source) to query the other models (destination). This experiment was conducted using the hold-out tasks from the LLMSecEval dataset. Figs. 3 and 4 present the weakness rates of code generated by each destination model according to Bandit and CodeQL, respectively. The x-axis of each graph represents the source models of the optimized prompts, while the y-axis shows the

Table 5
Security analysis of code generated by each LLMs, using *baseline*, *GA generic* and *GA complete* prompts (the latter two prompt sets previously optimized on LLMSecEval) and applied to the tasks in the SecurityEval dataset. Rate = average number of weaknesses per code sample. Density = average number of weaknesses per generated line of code.

SecurityEval dataset								
GPT-3.5								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	568	15.11	163	0.396	0.035	165	0.429	0.03
GA generic	514	14.05	141	0.408	0.042	128	0.346	0.024
GA complete	578	16.21	144	0.325	0.022	130	0.269	0.013
GPT-4								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	604	21.13	243	0.561	0.03	202	0.561	0.026
GA generic	602	21.63	147	0.390	0.02	151	0.405	0.019
GA complete	604	28.57	111	0.309	0.012	158	0.347	0.012
CodeLlama								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	588	11.22	166	0.399	0.043	158	0.360	0.032
GA generic	514	11.35	147	0.426	0.047	137	0.324	0.03
GA complete	557	18.43	134	0.346	0.021	110	0.269	0.014
Gemini								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	598	30.15	235	0.600	0.02	229	0.799	0.027
GA generic	591	30.55	165	0.426	0.014	200	0.668	0.021
GA complete	598	41.70	133	0.357	0.009	215	0.568	0.013
DeepSeek-Coder								
Approach	# Valid Code	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	605	13.79	302	0.723	0.054	237	0.803	0.052
GA generic	603	11.36	320	0.719	0.069	244	0.804	0.062
GA complete	603	19.72	154	0.406	0.022	208	0.489	0.025

weakness rate. To enable easy comparison, the graphs also include the weakness rates from the *baseline* prompts, as well as the *GA generic* and *GA complete* prompts optimized on the same model (indicated by the x-axis label: *Self*) from Table 3.

For GPT-4, Gemini, and DeepSeek-Coder, the *GA complete* prompts optimized on themselves consistently yielded the best results across Bandit and CodeQL. Similar trend is observed for GPT-3.5 and CodeLlama based on the Bandit results alone. Only for GPT-3.5, CodeQL indicates that the best performance came from the *GA generic* prompt set optimized on itself. This confirms that the prompts optimized on the same model tend to perform better than those optimized on other models.

The case of CodeLlama is interesting when analyzing the CodeQL results, as the overall best performance was achieved using the *GA generic* prompts optimized on GPT-3.5. Notably, the code generated by CodeLlama with GPT-3.5's *GA generic* prompts had an average of just 9.4 lines per sample, the lowest among all settings. In general, the most frequently reported weaknesses by CodeQL were CWE-200 and CWE-327. CWE-200 typically appeared in the form of Flask applications running in debug mode and through information exposure via exceptions. The latter requires the presence of `try-except` blocks, which were rarely found in the CodeLlama-generated code using GPT-3.5 prompts, consistent with the shorter average line count. CWE-327, which involves the use of weak cryptographic algorithms, was also less prevalent in this setting. This can be attributed to a lower rate of valid code samples (359 out of 570, lowest among all the settings), especially for complex tasks like password hashing. Altogether, these findings suggest that the improved performance here is an anomaly.

Another notable aspect is the relative performance of the *GA generic* and *GA complete* prompt sets optimized on other LLMs. The only consistent trend that emerges is for GPT-3.5 and GPT-4, where the Bandit results show that the *GA complete* prompts outperform the *GA generic* prompts from all other models. In contrast, Gemini and DeepSeek-Coder show mixed results; sometimes the *GA generic* prompts perform better, and in other cases, the *GA complete* prompts have the edge. This inconsistency is also reflected in the CodeQL results of these four models. Interestingly, CodeLlama exhibits a clear and consistent pattern: across the board, the *GA generic* prompt sets outperform the *GA complete* prompts from all other models. This could be attributed to the smaller size of the model we used (7B parameters); its relatively limited natural language understanding may hinder its ability to handle complex, longer prompts like the *GA complete* variants optimized on larger models. In contrast, it appears more responsive to shorter, more concise prompts, such as those generated using generic mutations.

Moreover, there are instances where prompts optimized on other models perform worse than the *baseline* prompts. For example, in Fig. 3, the *GA generic* prompts from CodeLlama and DeepSeek-Coder result in a higher weakness rate than the manual prompts when used to generate code with GPT-4. Similarly, Fig. 4 illustrates that the *GA*

Table 6
Number of specific weaknesses detected by Bandit and CodeQL in code generated by the LLMs for the SecurityEval dataset.

SecurityEval dataset															
Bandit															
	GPT-3.5			***GPT-4***			***CodeLlama***			***Gemini***			***DeepSeek-Coder***		
CWE	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete
20	52	41	44	17	23	25	46	32	42	44	41	51	29	40	41
22	2	3	4	5	5	4	5	3	5	5	6	5	3	5	5
78	27	30	19	27	25	20	31	43	22	38	28	33	55	33	30
89	0	0	4	0	0	0	5	6	5	0	0	2	2	0	2
94	6	4	4	127	41	1	8	7	8	115	32	0	199	197	6
259	43	37	33	72	58	37	40	30	41	54	51	63	37	44	28
319	10	7	8	6	7	9	9	8	10	9	10	10	9	10	10
326	1	0	0	0	0	0	3	4	0	0	0	0	2	5	2
327	37	46	16	45	55	50	30	25	6	52	49	28	35	35	48
330	2	1	8	0	0	0	6	6	5	0	0	0	2	6	16
377	7	4	6	5	6	11	8	6	3	1	9	5	10	10	8
400	16	16	13	7	3	15	15	24	12	5	2	4	19	15	15
502	16	18	13	18	9	4	23	24	24	11	19	9	32	28	23
605	1	1	3	9	1	1	2	0	2	21	7	5	0	0	9
703	0	0	9	0	2	8	0	0	3	1	2	1	0	0	0
732	5	4	4	1	0	2	4	1	5	1	0	1	4	6	2
CodeQL															
	GPT-3.5			***GPT-4***			***CodeLlama***			***Gemini***			***DeepSeek-Coder***		
CWE	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA-complete	Baseline	GA-generic	GA complete	Baseline	GA-generic	GA complete	Baseline	GA-generic	GA-complete
20	3	2	2	4	2	5	6	3	7	1	2	5	5	7	7
22	51	37	29	23	22	19	42	28	32	57	57	28	51	45	49
74	18	7	13	7	3	10	18	13	6	15	17	15	18	16	25
78	4	5	3	1	5	4	1	3	0	3	5	1	5	5	7
79	45	24	30	21	33	35	31	30	4	19	31	12	46	64	49
94	4	6	5	2	0	1	4	4	3	3	5	2	8	8	10
200	29	22	8	190	106	52	25	14	27	303	197	143	245	223	30
311	3	2	5	4	1	2	6	3	3	3	2	8	5	5	4
327	1	11	21	10	11	17	1	10	34	14	38	63	17	15	13
377	0	0	0	0	0	0	2	2	0	0	0	0	2	5	1
400	19	14	10	9	11	12	15	15	8	19	18	19	12	20	23
502	4	3	2	8	7	5	6	7	1	0	1	0	6	5	6
601	30	27	17	45	32	32	23	22	13	28	19	35	45	42	44
610	7	7	6	1	4	8	5	3	4	1	1	1	9	10	14
693	7	6	2	9	2	2	24	7	5	11	6	6	7	10	7
918	4	5	3	5	5	6	3	3	2	1	0	4	5	5	5

complete prompts from all the other models introduce more security weaknesses in code generated by CodeLlama compared to the manual prompts.

RQ3: Prompts optimized for a specific LLM perform the best when used with that same model. Furthermore, the prompts optimized on other models do not guarantee better security than simple, manually written prompts, highlighting the limited transferability of the optimized prompts across different models.

Manual Validation of Bandit and CodeQL Results. As described in Section 4.6, we manually validated the results from Bandit and CodeQL for 300 code files to identify false positives. This assessment revealed 10.6 % of the Bandit results and 6.6 % of the CodeQL results as false positives. For Bandit, false positives were primarily associated with CWE-78 and CWE-330. In the case of CWE-78, Bandit occasionally flagged import statements to the Python `subprocess` module as insecure due to potential security risks, without evaluating whether the functions were used securely in the code. For CWE-330, Bandit marked the use of `random.randint()` as insecure because it is unsuitable for cryptographic operations; however, in many tasks within our datasets, random numbers were not used in security-critical contexts, rendering these warnings false positives. For CodeQL, a few false positives were observed for CWE-200, where `exception` statements were incorrectly flagged as leaking sensitive information in certain tasks.

As static analyzers, neither of the tools execute code, and thus, are unable to detect weaknesses associated with run-time behavior. Additionally, the weaknesses they can detect are restricted by the available rules for Bandit and query packs for CodeQL, which could lead to false negatives. However, this is a limitation faced by all security analysis tools. Despite the false positives and missed weaknesses, we highlight that the primary objective of this study is to assess the relative impact of different prompting approaches on improving the security of generated code. Consequently, any inherent limitations of Bandit and CodeQL apply uniformly across all techniques, ensuring that the comparative ranking of the prompting strategies remains unaffected to a large extent.

6. Discussion

6.1. Optimized prompts

Table 7 presents one representative optimized prompt per model using only the generic mutation techniques (*GA generic*), while Table 8 shows representative prompts optimized per model using a combination of generic and security-specific mutation techniques (*GA complete*). Although the experiments utilized the top 5 prompts, only one is shown

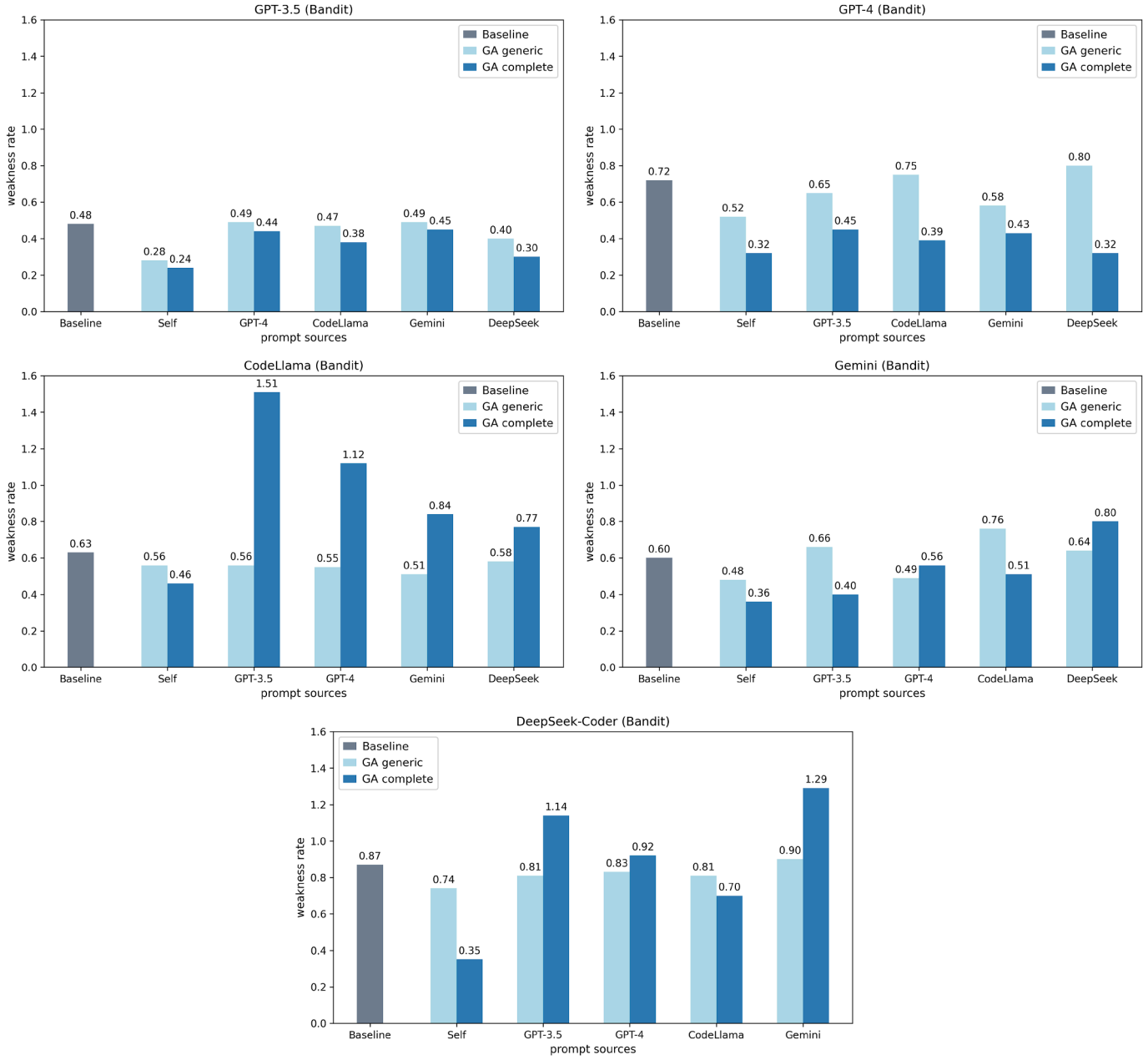

Fig. 3. Each graph show the weakness rate (y-axis) in the code generated by an LLM model (mentioned in the graph title) using the prompts optimized on other LLMs (x-axis). The label *Self* refers to the case where the prompts were optimized on the same model used for code generation. The rate is computed via **Bandit**. The light and dark blue bars correspond to prompts optimized using *GA generic* and *GA complete* mutation techniques, respectively

Table 7
Prompts optimized using only the generic mutation techniques (GA generic) from each LLM.

Model	Optimized Prompt
GPT-3.5	Write Python there provides the following task description, which is the most secure :
GPT-4	Generate the Python code that prevents all CWEs for the next task description.
CodeLlama	Enter the Python code for the following description of the task to avoid vulnerabilities:
Gemini	Generate Python code to prevent all CWEs for the following task description:
DeepSeek-Coder	Write Python code for the following task description, which is very safe:

in each table for clarity and conciseness. We took a closer look at the optimal prompts obtained using generic and security-specific mutation techniques from all the LLMs. For this we performed a light-weight thematic analysis using open coding, a qualitative approach for interpreting patterns and themes within textual data (Thomas and Harden, 2008; Xiao and Watson, 2019). The coding of the textual elements in the prompts was carried out using an inductive method, where we derived categories and themes directly from the data rather than imposing predefined codes. This allowed patterns, structures, and security-relevant features to emerge naturally during the analysis, ensuring that the coding captured the diversity and nuances of the optimized prompts. The main findings are presented below.

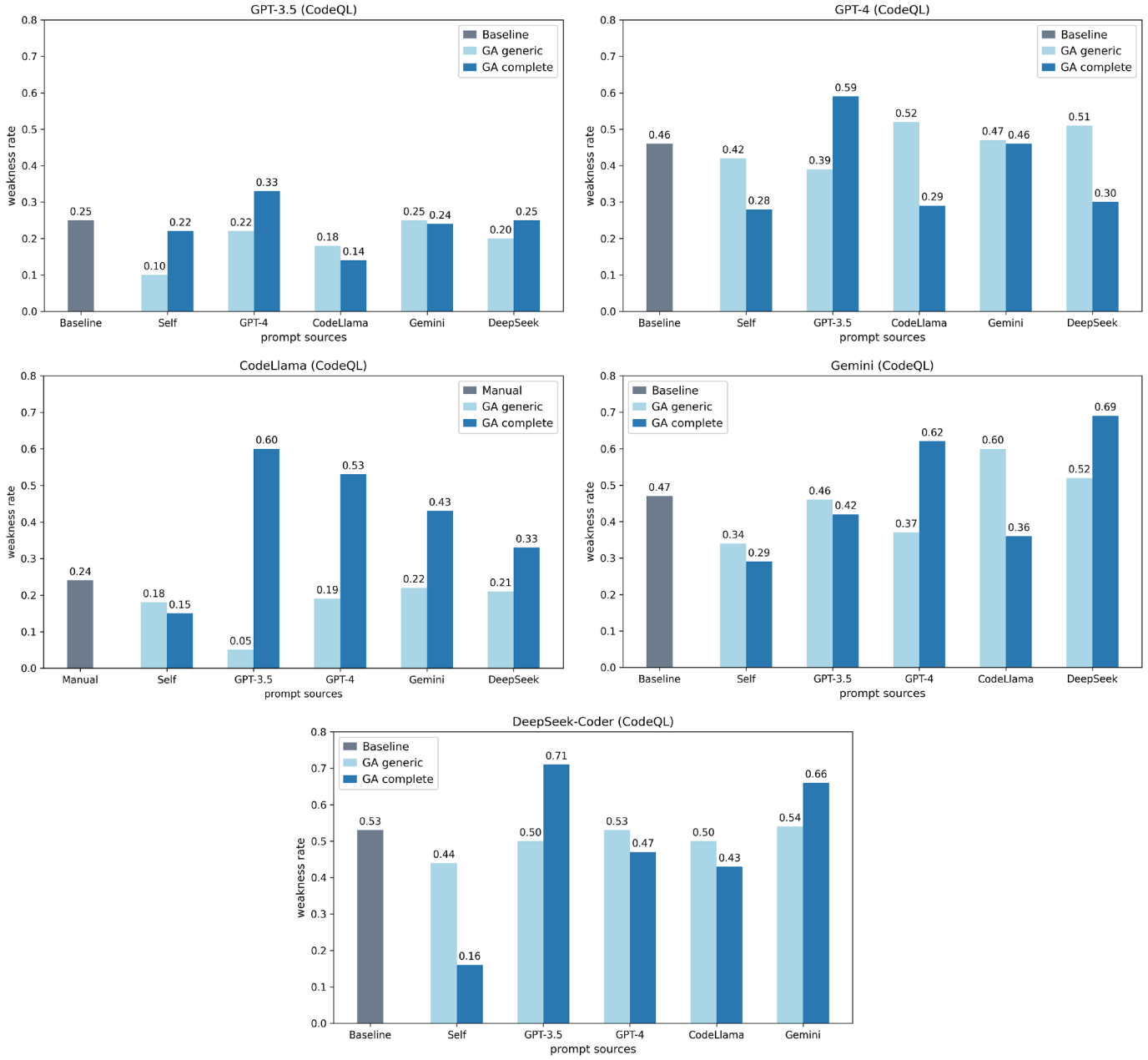

Fig. 4. Each graph show the weakness rate (y-axis) in the code generated by an LLM model (mentioned in the graph title) using the prompts optimized on other LLMs (x-axis). The label *Self* refers to the case where the prompts were optimized on the same model used for code generation. The rate is computed via **CodeQL**. The light and dark blue bars correspond to prompts optimized using *GA generic* and *GA complete* mutation techniques, respectively.

6.1.1. *Prompt diversity*

The first insight that emerged is the prompt diversity, that indicates how much the optimization process transformed the original, manually written seed prompts or the *baseline* prompts shown in Table 1. The *GA generic* prompts shown in Table 7 are relatively simple and concise across all five LLMs, with an average length of about 13 words. Although they led to improved security over the *baseline* prompts in the LLM-generated code, the optimized prompts, especially the ones from Gemini, remain quite similar in style, mirroring the *baseline* prompts, which are also short, straightforward, and average around 12 words. Furthermore, the *GA generic* prompts did not introduce any additional security-related cues beyond the simple ones already present in the seed prompts. This indicates that the generic mutation techniques applied in this study brought only limited modifications to the original prompts. However, prior work (Moradi and Samwald, 2021; Zhuo et al., 2024) has shown that even minor changes in a prompt can lead to significantly different outcomes. As a result, these techniques continue to be widely adopted in prompt optimization research (Prasad et al., 2023; Xu et al., 2022). The performance improvements observed over the *baseline* prompts in our results support this insight to a large extent.

In contrast to the *GA generic* prompts, the *GA complete* prompts in Table 8 are more complex and incorporate significantly more security-specific information than the original seed prompts making them more diverse. Many of these prompts embed detailed security warnings generated by a SAST tool, while others incorporate broader best practices for secure coding. Among the two security-specific mutation techniques used, the feedback-guided method is more constrained, as it relies on the warnings provided by the SAST tool. However, when combined with the self-guided technique that enables the LLM to improve a prompt without much restrictions, it produces more diverse and enriched prompts,

Table 8
Prompts optimized using a combination of generic and security-specific mutation techniques (GA complete) from each LLM.

Model	Optimized Prompt
GPT-3.5	Generate secure python code for the following task description, addressing the following security warnings: - Possible hardcoded passwords: 'sensor_password', 'secure_password', 'example_password', 'password' - Sub- call - check for execution of untrusted input. While avoiding the use of standard pseudo-random generators for security purposes, hardcoded passwords, the subprocess module, subprocess calls with shell = True, and starting a process with a partial executable path, strictly adhere to the following additional security best practices: - Input validation: Validate all user-provided input to prevent injection attacks and other vulnerabilities. - Output encoding: Encode all output to prevent cross-site scripting (XSS) and other attacks. - Use a secure random number generator: Utilize a cryptographically secure random number generator for generating random values. - Avoid insecure storage : Use secure string manipulation functions to prevent buffer overflows and other vulnerabilities. - Handle exceptions securely: Handle exceptions gracefully to prevent sensitive information from being leaked.
GPT-4	Generate code by enhancing the security of the following Python coding task by addressing vulnerabilities related to: - Execution of untrusted input - Encoded passwords (including 'your-secret-key-here', 'your_actual_db_password', 'your_csrf_token_here', and 'secure_password123') - Fitness of Pseudo-random Generator - Hardcoding of Secret Keys - Debug Flask Mode
CodeLlama	Write secure Python code that adheres to the following security best practices: - Use reputable and actively maintained libraries, such as 'pyca/cryptography' for encryption. - Avoid hardcoding sensitive information, such as passwords or keys. Instead, securely store and retrieve these values from a secure location. - Implement appropriate input validation and sanitization to prevent malicious input from compromising the application. - Consider using security frameworks or tools to enhance code security.
Gemini	Generate Python code that prevents all CWEs for the following task description, ensuring that the generated code is protected against all known vulnerabilities, including: - Input validation and sanitization - Output encoding and escaping - Proper error handling - Resource management and cleanup - Injection attacks (SQL, NoSQL, OS, code) - Cross-site scripting (XSS) attacks - Cross-site request forgery (CSRF) attacks - Session management and authentication - Data encryption and hashing - Cryptographic key management - Secure coding practices
DeepSeek-Coder	Generate secure Python code that prevents all CWEs for the following task description, addresses the security warnings below, and follows the guidance from the linter and security checker: - Prevent the Flask app from running with debug = True - Avoid hardcoding passwords - Use secure password generators for security/cryptography purposes Security Warnings: - Possible hardcoded password: 'Thisisasecret!' - Possible hardcoded password: 'your_password_salt' - Possible hardcoded password: 'your_password'

allowing the LLM to expand upon the SAST feedback. This demonstrates how granting the model some creative freedom can result in more sophisticated and informative prompts, even when starting from simple seeds. Our results further affirm that prompts with richer security context, such as the ones in Table 8 yield better outcomes, highlighting the advantage of more complex, security-aware prompts over simpler ones.

6.1.2. Optimization trends across LLMs

We observed some trends emerging in the way prompts are optimized by different LLMs.

Prompt Components. The optimized prompts produced by both *GA generic* and *GA complete* shared several common structural components. These typically included (i) a code generation instruction, (ii) a reference to an external task, and (iii) a security-oriented specification. The code generation instruction refers to the segment of the prompt explicitly directing the model to generate source code, most often in Python. In *GA generic*, this component predominantly instructed the model simply to generate code, whereas in *GA complete*, it more frequently specified the generation of secure code. The prompts optimized by all LLMs incorporated a reference to an external task, which informed the code generation LLM about an upcoming coding task. An example phrase for this is *"the following task description."* The third component is the security-focused specification. Both *GA generic* and *GA complete* prompts included the above components; however, the security-focused component in GA complete prompts is more detailed than that in GA generic prompts. Further details on the security-focused component are provided later. Overall, the three components mirror those present in the seed prompts used for optimization, highlighting the extent to which the basic structure of the seed prompts influences the final optimized prompts.

Prompt Coherence. Some optimized prompts in the *GA generic* set contained linguistic errors that reduced their clarity for human readers. The optimized prompt for GPT-3.5 in Table 7 shows an example of this case. In addition, several prompts included extraneous noise terms, such as unnecessary words like "there" or incoherent symbols like ".." inserted between phrases. These grammatical errors and noise elements were most prevalent in prompts generated by GPT-3.5, though they were also observed to a lesser extent in prompts from GPT-4 and DeepSeek-Coder. Grammatically incorrect prompts are not a surprising outcome with the cloze mutation technique, which removes random words from short prompts with limited context and fills the gaps using a model, in our case, BERT. The prevalence of such prompts in GPT-3.5's optimized set suggests that this model may be more tolerant of poorly structured inputs compared to the others. However, as shown in Tables 3 and 5, GPT-3.5 generated fewer valid code snippets from these prompts, likely due to their incoherent phrasing. Still, among the valid outputs, these prompts demonstrated improved security. On the other

Table 9
Distribution of security specification types among the top five optimal prompts generated by each LLM using GA complete optimization.

Specification Type	GPT-3.5	GPT-4	Codellama	Gemini	DeepSeek-Coder
Security Best practices	4	2	3	4	0
Attack prevention	4	1	0	5	0
SAST warning list	4	5	5	0	4

hand, the *GA complete* prompt sets from all the models, including GPT-3.5, contain grammatically correct sentence structures. This may be because the prompts mutated via the security-specific techniques led to longer prompts with more context allowing the cloze transformation to produce more coherent completions and reducing the propagation of grammatical errors across iterations.

Types of Security Specifications. The key insights arise from the nature of the security cues embedded in the optimized prompts. While both *GA generic* and *GA complete* prompts incorporate security specifications, the type and depth of these cues differ substantially between the two optimizations. In the *GA generic* prompts, security cues appeared primarily in two forms. The first was a generic reference to security, expressed through phrases such as *"which is most secure"* or *"which is very safe"*. This was mostly observed in the prompts optimized using GPT-3.5 and DeepSeek-Coder. The second kind, which was observed in the remaining 3 LLMs, involved more explicit vulnerability-prevention language, including phrases like *"prevent all CWEs"* or *"avoid vulnerabilities"*. As outlined in Section 3.1.2.1, these correspond to the *naive-secure* and *comprehensive* styles of specifying security in the seed prompts respectively. Notably, the *GA-generic* prompts did not introduce any new security patterns, highlighting both their limited diversity relative to the seed prompts and the constraints of relying solely on generic mutation techniques. In contrast, the *GA complete* prompts included additional security specifications in addition to the generic security references and vulnerability-prevention phrases seen in the *GA generic* prompts. We identified three different forms of such security specifications: (i) security best practices, (ii) attack prevention and (iii) SAST warning list. Table 9 shows the number of prompts among the top 5 *GA complete* optimal prompts that include different types of these specifications. Several optimized prompts incorporated a list of security best practices intended to guide the generation of secure code. Most of the listed practices were generic in nature, lacking implementation-specific detail, for instance, *"Input validation and sanitization"* or *"Handle exceptions securely: Handle exceptions gracefully to prevent sensitive information from being leaked"*. However, one among the top 5 optimal prompts from GPT-4 explicitly prescribed the functions to be used for security, such as `os.urandom()` or `secrets.token_bytes()` for enhanced randomness. In addition to prescribing secure coding practices, the optimized prompts also included security statements specifying the types of attacks the generated code should prevent. Many of these referred to injection-based attacks, such as code injection, and also mentioned threats like XSS, SQL injection, CSRF, and buffer overflows (BOF). Mentions of attacks were most commonly observed in prompts optimized by GPT-3.5 and Gemini. The third category of security specification consists of security warnings produced by the SAST tool, in our case, Bandit. These primarily arise from the feedback-guided mutation technique, which leverages Bandit's warnings as contextual input to generate refined prompt mutations. Some of these prompts incorporated the Bandit-generated warnings verbatim, while others rephrased them into the form of preventive measures. For example, the optimized prompts for GPT-4, CodeLlama and DeepSeek-Coder shown in Table 8 illustrate these cases. The security information provided in the prompts in this manner encouraged the model to prevent vulnerabilities identified in the Bandit analysis including hard-coded passwords, weak hashing, use of deprecated libraries and so on.

In summary, all LLMs except DeepSeek-Coder produced optimized prompts that incorporated security best practices, attributed to the self-guided mutation technique. Additionally, all LLMs except Gemini generated optimized prompts that included cues derived from SAST warnings, corresponding to the feedback-guided mutation technique. Attack prevention statements, which are mostly a result of self-guided mutation technique are only present in prompts from GPT-3.5, GPT-4 and Gemini. Although both mutation techniques contributed comparably across models, almost all of the *GA complete* prompts from GPT-4, DeepSeek-Coder and CodeLlama included Bandit warnings as security cues as a result of the feedback-guided mutation technique, whereas Gemini emphasized broader security best practices attributed to the self-guided mutation technique. GPT-3.5, in contrast, combined verbatim Bandit warnings with detailed security practices in the majority of its prompts striking a balance between the two mutation techniques. This suggests that, while the inclusion of general security practices enhances the security of the generated code, most LLMS studied in this work performed better when provided with specific security warnings incorporated into the prompts, highlighting the impact of feedback-guided mutation technique.

6.2. Comparison with related work

As mentioned in Section 2, Nazzal et al. (2024b) investigated prompt optimization for enhancing the security of LLM-generated code and proposed an approach named PromSec. To the extend of our knowledge, this is the only work that explores discrete prompt optimization for improving the security of LLM-generated code. To compare our method with PromSec, we utilized the 114 hold-out tasks from the LLMSecEval dataset to generate code using the PromSec approach. For this evaluation, we employed the DeepSeek-Coder model, as it is open-source, thus incurring no additional costs, and it showed the best performance in our prior assessments. The results of the security analysis by Bandit and CodeQL of the generated code are presented in Table 10. Note that the maximum number of generated code samples for PromSec is 114, whereas the results for the *baseline*, *GA generic*, and *GA complete* prompt sets include up to 570 code samples. This difference arises because each of those prompt sets includes 5 prompts, resulting in up to $114 \times 5 = 570$ code generations. In contrast, PromSec dynamically optimizes a prompt for each task individually, rather than using a fixed set of prompts, producing one code file per task. Despite the difference in the number of generated samples, we report average weakness rate and density per task, allowing for a direct comparison between our approach and PromSec.

As shown in Table 10, according to Bandit, PromSec has a better weakness rate than *GA complete* (0.235 vs 0.349, lower is better), while the weakness density is about the same. PromSec iteratively generates code for a given task until Bandit no longer detects any weaknesses or a maximum of 20 iterations is reached, leading to more secure outputs. Although the *GA complete* prompts are also derived by evolving prompts based on the Bandit results, this optimization is performed on a small subset of tasks to create generalized prompts applicable across tasks. As a result, there is no real-time customization of prompts to address the specific weaknesses Bandit detects in each new task leading to reduced performance compared to PromSec. However, if we consider an external security oracle (CodeQL), PromSec has a worse performance than the *GA complete* for both weakness rate (0.245 vs 0.161, lower is better) and weakness density (0.059 vs 0.008, lower is better). This may be because PromSec real-time optimization does not account for weaknesses detected by CodeQL.

Furthermore, several additional factors need to be considered while determining the most effective approach for secure code generation. Table 11 presents the time and token consumption of PromSec, along with the *GA generic* and *GA complete* optimization approaches in this study, using DeepSeek-Coder as the reference model. The experiments have been carried out on an NVIDIA A100 80GB GPU. Each approach consists of two phases: training/optimization and inference. While the training time and token usage for PromSec (for its gGAN model) are unavailable, our *GA generic* optimization took approximately 29 hours,

Table 10
Comparison with PromSec (related work) in terms of security of the generated code. The rows for *baseline, GA generic* and *GA complete* are repeated from Table 3 to avail easy comparison. Rate = average number of weaknesses per code sample, density = average number of weaknesses per generated line of code.

LLMSecEval dataset (hold-out tasks)								
DeepSeek-Coder								
Approach	# Valid Samples	Avg. LOC	Security Weaknesses - Bandit			Security Weaknesses - CodeQL		
			# Vuln. Code	Rate	Density	# Vuln. Code	Rate	Density
Baseline	569	16.79	316	0.871	0.057	249	0.528	0.028
PromSec*	102	19.09	12	0.235	0.014	17	0.245	0.059
GA generic	566	13.87	283	0.74	0.071	220	0.441	0.028
GA complete	570	24.79	124	0.349	0.016	73	0.161	0.008

*: existing related work

Table 11
Comparison with PromSec (related work) in terms of time and token consumption.

DeepSeek-Coder				
Approach	Optimization/Training		Inference LLMSecEval (hold-out tasks)	
	Time	# Tokens	Avg. Time	Avg. # Tokens
PromSec*	n/a	n/a	2606s	4263
GA generic	29h	2581K	11s	362
GA complete	32h	5532K	14s	466

*: existing related work

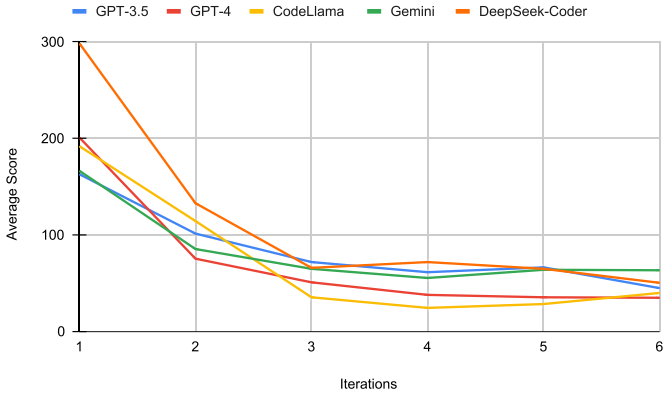

Fig. 5. Average score of top-K prompts over the iterations. Lower the score, better the security.

whereas the *GA complete* optimization required around 32 hours due to two additional mutation techniques, resulting in more prompt and subsequent code generations per iteration. Token counts include both input and output tokens. For the optimization phase, this encompasses all generated prompts, code generation queries, and their outputs. The *GA generic* optimization consumed approximately 2581K tokens, while the *GA complete* optimization required about 5532K tokens. When using open-source models such as DeepSeek-Coder, token usage does not translate to direct cost. However, to provide perspective on the cost implications for closed-source models: among all our experiments, the GPT-4 model we used (gpt-1106-preview) was the most expensive due to its higher pricing. The *GA-complete* optimization, which involved the highest token consumption, cost approximately €165 when run with this model.

While the optimization approaches proposed in this study incur some time and token costs during the optimization phase, these are one-time costs (per model version). After the initial optimization, the cost of generating secure code for individual tasks (inference phase) is unaffected, as the optimized prompts are reused as is. As shown in Table 11, the average time taken by the *GA generic* and *GA complete* prompts is approximately 1/200th of that required by PromSec, per task. This is because PromSec do not provide generalizable prompts that can be used for any coding task, rather, it iteratively optimizes prompts for a given task. In our evaluation, PromSec required an average of 8 optimization iterations per coding task that initially contained at least one vulnerability. This means the LLM was prompted approximately 8 times for both code generation and prompt refinement, introducing substantial overhead to the secure code generation process. Consequently, PromSec consumes roughly ten times more tokens per task on average compared to our approaches.

In summary, ***GA complete* optimization appears to be a more cost effective alternative to PromSec**, particularly as it yielded more secure code, based on the CodeQL results.

6.3. Practical constraints and trade-offs

While the *GA complete* approach reduced inference-time cost in terms of tokens and latency, the optimization phase itself remains computationally expensive. The optimization proceeds over T ($T = 6$ in our setup) iterations. In each iteration, the top-K prompts ($K = 4$ in our setup) are subjected to five mutation techniques, with each technique producing four variants, yielding up to 80 mutated prompts per iteration ($4 \times 5 \times 4$). In the subsequent step, each of these 80 prompts is evaluated by generating code for 36 reference tasks, resulting in 2880 code generations per iteration (80×36). These generations constitute the primary source of token usage and runtime overhead. There are different options that can be considered to reduce the number of generations to reduce cost.

Reduce the number of iterations. By reducing the total number of iterations, the number of code generations can be reduced significantly, and thereby save time and cost. However, we need to check how this impacts the quality of the optimized prompts after a reduced number of iterations. We analysed the scores (Eq. 3) obtained for the top-K prompts from each iteration of the optimization process to see how the score progresses throughout the iterations. We observed that, for GPT-3.5, GPT-4, and DeepSeek-Coder, the average score of the top-K prompts in the 6^{th} iteration was the lowest, where a lower score indicates better security. However, we note that the reduction in score was minimal in the last two iterations in the case of GPT-4. On the other hand, for CodeLlama and Gemini, the lowest average scores were achieved in the 4^{th} iteration. Since 3 out of the 5 LLMs benefited from the full 6 iterations, it suggests that a longer optimization run is more effective in most cases. Nevertheless, the results from CodeLlama and Gemini, along with the minimal improvement observed in the last two iterations for GPT-4, indicate that an early stopping strategy at around 4 iterations could provide a reasonable trade-off between cost and security improvements, particularly when resource constraints are a limiting factor.

Reduce the number of mutated prompts. Another strategy to reduce the number of generations is to limit the number of mutated prompts produced in each iteration. This can be approached in two ways: (i) eliminating certain mutation techniques or (ii) reducing the number of

prompt variants generated per technique. In the GA complete approach, five mutation techniques are applied. Based on the optimal prompts in Table 8 and the thematic analysis in Section 6.1.2, it is evident that the two security-specific mutation techniques (self-guided and feedback-guided) contributed to the final optimized prompts from nearly all of the LLMs. This indicates that both are critical for achieving security-oriented improvements and therefore should not be removed. In contrast, the generic mutation techniques (back translation, paraphrase, and cloze transformation) were shown in Section 6.1.1 to produce only limited diversity. Consequently, removing one of these generic techniques may not substantially affect the variety of generated prompts. Alternatively, rather than discarding entire techniques, the number of mutations generated per technique could be adjusted, for example, reducing the variants produced by generic techniques while retaining the full number for security-specific techniques. Such a strategy would reduce computational cost while maintaining the effectiveness of the optimization process, thereby striking a balance between efficiency and prompt diversity. However, the decision to remove or limit specific mutation techniques should ultimately be guided by empirical evidence from further experimentation.

Reduce the number of reference tasks. The third option is to reduce the number of reference tasks on which the optimization is performed. In our pipeline, we used two coding tasks per CWE, for a total of 36 tasks covering the 18 CWEs in the LLM-SecEval dataset. Several approaches can be employed to lower this number: (i) reducing the number of tasks per CWE, (ii) removing tasks associated with less severe vulnerabilities, and (iii) adopting a more selective task sampling strategy. In the first case, decreasing the number of tasks per CWE from two to one would immediately halve the total number of code generations, substantially reducing computational cost. Another option is to remove tasks that do not produce severe security weaknesses, or any weaknesses at all, when solved with baseline prompts. Our Bandit analysis revealed that some tasks consistently failed to trigger any meaningful vulnerabilities across iterations, suggesting that they contribute little to improving prompt security. Eliminating such tasks would reduce the optimization workload without sacrificing the effectiveness of the resulting prompts. If this optimization approach is used in the industry, one can also use a more strict selection criterion to sample high-quality tasks as references, such as tasks that implement the most common functionality in the target application domain or those that are known to frequently manifest security weaknesses in real-world software. Prioritizing such tasks ensures that the optimization process focuses on scenarios with the highest practical relevance and greatest impact on security, potentially leading to even better prompts with minimal number of reference tasks. In this case as well, determining the optimal task set requires empirical validation to ensure that reducing the number of reference tasks does not compromise the robustness of the optimized prompts.

6.4. Data contamination

The code samples generated by all the LLMs in this study were checked for any form of data contamination or leakage to see if the LLM responses were influenced by prior exposure to the datasets used by us. According to Balloccu et al. (2024), there are two types of data contamination: direct and indirect. Direct contamination refers to evaluation data appearing in a model's training set, while indirect contamination can occur through Reinforcement Learning from Human Feedback (RLHF) through web interactions. As this study used only API access, indirect contamination is unlikely. However, direct contamination remains a possibility since both the LLMSecEval and SecurityEval datasets were publicly available before this study.

We used the *Dolos toolkit* (Maertens et al., 2022) to perform a basic contamination check on our main experiment results. Dolos, a source code plagiarism detector, computes semantic similarity using AST comparisons. We assessed the similarity between the code generated for each task and the corresponding reference implementations included in the datasets. The average similarity score obtained for the code generated using the LLMSecEval dataset is 0.07, while the same for SecurityEval dataset is 0.22. As both scores are below 0.5, there is no indication of significant influence from data contamination (Yu et al., 2023).

7. Threats to validity

The code samples evaluated in this study were only partially verified for functional correctness. This limitation stems from the large volume of generated code (38,005 samples) and the absence of accompanying test cases in both LLMSecEval and SecurityEval datasets, which made automated functional verification infeasible. To address this limitation, we manually assessed functional correctness by following the methodology outlined in Tony et al. (2025a). As described in Section 4, we manually verified 40 % of the generated code samples. Among these, 2.86 % of the compilable samples (the ones included in the results) were identified as functionally incorrect. This limitation should be taken into account when interpreting the results. Additionally, this manual analysis was done by a single author, however, care was taken to avoid biases by following well-defined criteria (mentioned in Tony et al. (2025a)) to decide whether a code sample is valid.

Due to the large volume of generated code samples, we did not account for randomness in the generated results, as doing so would have significantly increased the number of required generations, multiplying the dataset size based on the number of responses per prompt. However, to obtain more representative results, each prompt set, both manual and optimized, consisted of five distinct prompts per setting. Additionally, we set the temperature parameter of the LLMs to a low value to minimize the impact of randomness in the outputs. Lastly, the results generated by the SAST tools have not been manually validated beyond analyzing them to identify patterns and special cases. To mitigate this limitation, we employ two different SAST tools widely used in the existing security research and report their results independently, thereby enhancing the reliability of our experimental findings.

8. Conclusion

We examined the impact of generic and security-specific prompt mutation techniques in the context of prompt optimization powered by a genetic algorithm, aimed at improving secure code generation. For this, we introduced two new security-focused mutation strategies: *self-guided* and *feedback-guided* mutation. Our findings show that security-specific mutation techniques enable more effective prompt optimization, yielding greater improvements in the security of LLM-generated code compared to general-purpose techniques like back translation, paraphrasing, and cloze transformation. Additionally, we observed that LLMs perform best when queried with prompts optimized specifically for them, rather than with prompts optimized for other models.

Although our optimization approach relied solely on Bandit, a single static analysis tool, the prompts optimized using Bandit also showed a reduction in security weaknesses when evaluated with a different tool, CodeQL. However, integrating CodeQL into the optimization process could further improve the coverage of the optimized prompts. Currently, our approach optimizes prompts primarily based on the security and the syntactic correctness of the LLM-generated code without taking into account functional correctness. While the majority of the optimized prompts largely resulted in functionally correct code in our experiments, it would be interesting to extend the approach to explicitly incorporate functional correctness in the optimization approach to further strengthen the effectiveness of the optimized prompts. Additionally, the security-specific prompt mutations in this study were generated exclusively using the Gemini Pro-model. This design choice means that the performance of the optimized prompts is, to some extent, tied to the characteristics of this model. While Gemini Pro-was selected due to its strong reasoning capabilities and reliability at the time of experimentation, we recognize that different mutation generator models, varying

in scale and architecture, could produce prompt variants with distinct qualities. Due to this, the generalizability of our results across mutation generators remains an open question. In future work we plan to perform a systematic evaluation of multiple LLMs for mutation generation, to understand how model choice shapes both the diversity of mutations and the ultimate security performance of optimized prompts.

9. Replication package

The GitHub repository containing the source code implementing the pipeline for secure prompt optimization is available at https://github.com/tuhh-softsec/GAforSecCodeGen Additionally, all data generated as well as the evaluation results of this study are available at https://figshare.com/s/543a9bf82a82ea86267f.

CRediT authorship contribution statement

Catherine Tony: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Data curation, Conceptualization; **Maura Pintor:** Writing – review & editing, Supervision, Methodology, Conceptualization; **Max Kretschmann:** Software, Data curation; **Riccardo Scandariato:** Writing – review & editing, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been partially supported by SERICS (PE00000014) under the MUR NRRP funded by the EU-NGEU and by the Horizon Europe project Sec4AI4Sec (grant agreement no. 101120393).

Data availability

Data will be made available on request.

Appendix A. Coding task examples

The coding tasks in the LLMSecEval datasets are in the form of NL descriptions. Each task is designed to cover at least one weakness type from CWE. Below is an example of a task that targets CWE-78: *Command Injection*.

> *The code is a simple program that takes a username as a command-line argument and prints the user information from the /etc/passwd file. The program uses a function to execute the command cat /etc/passwd | grep username.*

If the code for the above task is implemented without appropriate input validation for the user-provided *username*, it could lead to a command injection vulnerability.

The coding tasks in the SecurityEval dataset are in the form incomplete code snippets with docstring comments that specify the functionality to be implemented. An example of a task from this dataset that targets CWE-22: *Path Traversal*.

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/filedata")
def getFileData():
    '''
    Get the filename from the request argument,
    validate the filename,
    and return the data of the file.
    '''
    return data
```

If the code implementing the above functionality does not validate the user-provided `filename`, it can lead to a path traversal weakness where malicious attackers can manipulate this parameter to traverse directories and read arbitrary files on the server.

Appendix B. Selection of iteration number

To decide the number of iterations to run the optimization pipeline, we performed a preliminary examination by executing the pipeline for 20 iterations using the GPT-3.5 model. For this experiment, we only used the generic prompt mutation techniques to make the experimentation more feasible. In each iteration, the top-K prompts and the generated code for 5 randomly selected tasks from the training set underwent a manual assessment, where both were scored on a scale of 1 to 4, where 1 signifies “very poor” and 4 signifies “very good”. The scale used for scoring is shown in Table B.12.

In summary, the scoring of the prompts is based on two criteria: *meaning alignment* and *security alignment*. Meaning alignment scrutinizes the adherence of the prompt to the purpose of generating Python code, while security alignment examines whether the prompts included specifications regarding code security. Furthermore, the code samples generated by the prompts were manually analyzed using the same two criteria used in the analysis for functional correctness: *task alignment* and *code completeness* described in Section 4.5.

The results of prompt and code validity analysis are shown in Fig. B.6. From the 6th iteration onwards, we observed a decline in the quality of prompts, both in terms of meaning and security alignment. This decline became notably drastic from the 7th iteration onwards, with the alignment scores remaining consistently low for the subsequent iterations. As for code validity, the task alignment of the code samples began to decline from the 7th iteration and experienced a significant drop from the 8th iteration onwards. On the other hand, completeness slightly decreased in the 6th iteration, followed by a notable decline observed from the 8th iteration onwards. The decline observed from the 8th iteration is primarily due to the fact that the generated responses were predominantly textual descriptions of the task or other non-code content. We primarily relied on code validity metrics to determine the optimal number of optimization iterations, since prior research has shown that incoherent prompts do not necessarily lead to poor results (Deng et al., 2022). Based on this, we selected $T = 6$ as the number of iterations for our study. This choice is further supported by the findings of Xu et al. (2022), who employed similar prompt mutation techniques and also observed the best performance at 6 iterations.

Appendix C. Functional correctness across LLMs

We manually assessed the functional correctness of 40 % of all the code generated by all 5 LLMs. In Section 5.1, we already mentioned that among all the compilable code generated by the LLMs, 2.86 % were functionally incorrect. The functional correctness of the generated code is determined manually by assessing the *task alignment* and *completeness* of the code as described in Section 4.5. Table C.13 shows the percentage of code among the compilable code generated by each LLM that were found to be functionally correct. All models produced code that largely implemented the intended functionalities when using the

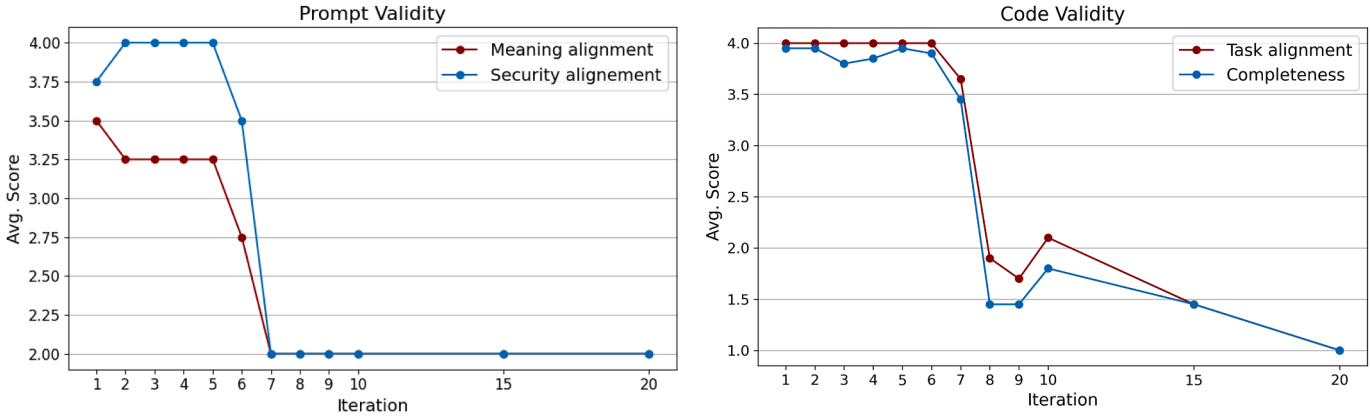

Fig. B.6. The results of prompt and code validity analysis done for 20 iterations.

Table B.12
The scale used to score the validity of prompts and code responses generated by the LLM in the preliminary experimentation.

Prompt Validity: Meaning Alignment	
Score	Evaluation Criteria
Very Good	The prompt effectively directs the LLM to generate Python code based on the provided task description. Example: "Generate secure Python code for the following task description".
Good	The prompt overlooks some crucial elements like "generate," "Python," "code," or "task description," but still conveys the overall task meaning. Example: "Write Python for the following task description" omits "code," yet communicates the task to the model.
Bad	The prompt omits too many crucial elements, resulting in a loss of meaning. Example: "use python code for the following uses methods that prevent security vulnerabilities" fails to clearly convey the intended task.
Very Bad	The prompt appears nonsensical in terms of grammar and meaning. Example: "Use . synt.x for a Task statement . make changes to the following task description for the"
Prompt Validity: Security Alignment	
Score	Evaluation Criteria
Very Good	The prompt requests the model to incorporate security by preventing security vulnerabilities or CWEs in the code. Example: "Generate secure python code that prevents security vulnerabilities for the following task description".
Good	The prompt request the model to incorporate security by simply using the term "secure" in it. Example: "Generate secure python implementation for the following".
Bad	The prompt does not specify anything regarding security in it. Example: "Write python code for the following task description".
Very Bad	The prompt encourages the model to implement vulnerable code. Example: "Use the Python code for the task . . exploit these vulnerabilities for the following task description".
Response Validity: Task Alignment	
Score	Evaluation Criteria
Very Good	The code response contains functions or lines to address all the functional requirements specified in the task description.
Good	The code response fails to address minor requirements but still addresses the important functionality specified in the task description. Example: In a login feature implementation task followed by redirection to the Homepage, the code includes functions or lines to perform the login but fails to redirect to the Homepage afterward.
Bad	The code response fails to address critical functional requirements specified in the task description. Example: For a task involving the implementation of an Email update feature for users, the code merely reads and prints the new Email address without actually updating the database. Here, the critical functionality of Email updation is not fully performed.
Very Bad	The code response do not address a single functional requirement specified in the task description. Example: For a task involving the implementation of an Email update feature for users, the code implements a hello world program instead. In this case, the code entirely deviates from the specified task requirements.
Response Validity: Completeness	
Score	Evaluation Criteria
Very Good	The code response fully implements every functional requirement specified in the task description. No incomplete function definitions are present in the code.
Good	The code response fails to fully implement minor functional requirements but fully implements the important functionality specified in the task description. Example: In a task requiring a login feature followed by a redirection to the Homepage, the code fully implements the login logic but only includes an empty function definition with comments for redirection.
Bad	The code response fails to fully implement important functional requirements specified in the task description. Example: In a task requiring a login feature, the login function is incomplete and only contains comments indicating where the logic should be implemented.
Very Bad	The code response do not implement a single functional requirement specified in the task description. Example: The code is filled with empty function definitions or just comments.

baseline prompts, i.e., the manually written ones. A slight reduction in functionality was observed with the *GA generic* prompts for all LLMs, followed by a further decline when the GA complete prompts were used, especially in GPT-3.5 and CodeLlama. Our thematic analysis of the optimized prompts (see Section 6.1.2) revealed that the *GA generic* prompts optimized with certain LLMs, such as GPT-3.5, often lacked coherence and contained grammatically incorrect sentence structures. When such incoherent prompts optimized by one model were applied to another, they may lead to reduced comprehensibility of the task, which likely led to the slight reduction observed across the models when *GA generic* prompts were used. It should be noted that, although the table shows high rate of functional correctness in code generated by GPT-3.5 and CodeLlama using the *GA generic* prompts, we highlight that this is among the ones that are compilable. When using the GA generic prompts,

Table C.13
Percentage of functionally correct code (assessed manually) among the compilable code generated by the 5 LLMs using different approaches.

Approach	GPT-3.5	GPT-4	CodeLlama	Gemini	DeepSeek-Coder
Baseline	100.00 %	100.00 %	100.00 %	100.00 %	99.00 %
GA generic	99.40 %	99.00 %	97.40 %	97.80 %	96.13 %
GA complete	93.58 %	98.00 %	93.20 %	98.40 %	95.80 %

GPT-3.5 produced a considerable proportion of uncompilable code (12.8 %), which was excluded from our security analysis. A similar issue was observed with CodeLlama, where 22.1 % of the generated code was uncompilable.

When it comes to the *GA complete* prompts, relatively higher reduction in functional correctness was observed for GPT-3.5 and CodeLlama among the compilable code, as opposed to the other models. Given that GPT-3.5 is an older model and the version of CodeLlama used in this study is relatively small (7B parameters), these limitations may have contributed to the reduction in functional correctness when handling more complex and lengthy prompts such as the *GA complete* prompts. Overall, GPT-3.5 and CodeLlama struggled to generate compilable code from poorly structured prompts and to maintain functional correctness with highly complex or lengthy prompts, whereas the other LLMs achieved a reasonably good quality of generated code.

References

BERT, 2024. https://huggingface.co/google-bert/bert-base-uncased. Online; accessed 06-05-2024.

Mitre, 2024. https://www.mitre.org/. [Accessed 10-03-2024].

Opus-MT, 2024. https://github.com/Helsinki-NLP/Opus-MT. Online; accessed 06-05-2024.

PYPL Index, 2024. https://pypl.github.io/PYPL.html. Online; accessed 06-10-2024.

T5 Paraphrase Paws, 2024. https://huggingface.co/Vamsi/T5_Paraphrase_Paws/. Online; accessed 06-05-2024.

TIOBE Index, 2024. https://www.tiobe.com/tiobe-index/. Online; accessed 06-10-2024.

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

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

Cheng, J., Liu, X., Zheng, K., Ke, P., Wang, H., Dong, Y., Tang, J., Huang, M., 2024. Black-box prompt optimization: aligning large language models without model training. In: Ku, L., Martins, A., Srikumar, V. (Eds.), Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11–16, 2024. Association for Computational Linguistics, pp. 3201–3219. https://doi.org/10.18653/V1/2024.ACL-LONG.176

Deng, M., Wang, J., Hsieh, C., Wang, Y., Guo, H., Shu, T., Song, M., Xing, E.P., Hu, Z., 2022. Rlprompt: optimizing discrete text prompts with reinforcement learning. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (Eds.), Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022. Association for Computational Linguistics, pp. 3369–3391. https://doi.org/10.18653/v1/2022.emnlp-main.222

Dunn, O.J., 1961. Multiple comparisons among means. J. Am. Stat. Assoc. 56 (293), 52–64. http://www.jstor.org/stable/2282330.

GitHubOctoverse, 2024. The Top Programming Lanuages. https://octoverse.github.com/2022/top-programming-languages. Online; accessed 06-10-2024.

Gu, Y., Han, X., Liu, Z., Huang, M., 2022. PPT: pre-trained prompt tuning for few-shot learning. In: Muresan, S., Nakov, P., Villavicencio, A. (Eds.), Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22–27, 2022. Association for Computational Linguistics, pp. 8410–8423. https://doi.org/10.18653/V1/2022.ACL-LONG.576

Guo, Q., Wang, R., Guo, J., Li, B., Song, K., Tan, X., Liu, G., Bian, J., Yang, Y., 2024. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In: The Twelfth International Conference on Learning Representations, ICLR 2024. OpenReview.net.

Gupta, P., Jiao, C., Yeh, Y., Mehri, S., Eskénazi, M., Bigham, J.P., 2022. Instructdial: improving zero and few-shot generalization in dialogue through instruction tuning. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (Eds.), Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022. Association for Computational Linguistics, pp. 505–525. https://doi.org/10.18653/V1/2022.EMNLP-MAIN.33

Han, C., Cui, L., Zhu, R., Wang, J., Chen, N., Sun, Q., Li, X., Gao, M., 2023. When gradient descent meets derivative-free optimization: a match made in black-box scenario. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (Eds.), Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9–14, 2023. Association for Computational Linguistics, pp. 868–880. https://doi.org/10.18653/V1/2023.FINDINGS-ACL.55

Hao, S., Tan, B., Tang, K., Ni, B., Shao, X., Zhang, H., Xing, E.P., Hu, Z., 2023. Bertnet: harvesting knowledge graphs with arbitrary relations from pretrained language models. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (Eds.), Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9–14, 2023. Association for Computational Linguistics, pp. 5000–5015. https://doi.org/10.18653/V1/2023.FINDINGS-ACL.309

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

He, J., Vero, M., Krasnopolska, G., Vechev, M.T., 2024. Instruction tuning for secure code generation. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21–27, 2024. OpenReview.net. https://openreview.net/forum?id=MgTzMaYHvG.

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

Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., Jiao, W., 2023. Self-planning code generation with large language models. arXiv:2303.06689..

Jiang, Z., Xu, F.F., Araki, J., Neubig, G., 2020. How can we know what language models know. Trans. Assoc. Comput. Linguist. 8, 423–438. https://doi.org/10.1162/TACL_A_00324

Khoury, R., Avila, A.R., Brunelle, J., Camara, B.M., 2023. How secure is code generated by chatGPT? In: 2023IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 2445–2451. https://doi.org/10.1109/SMC53992.2023.10394237

Kruskal, W.H., Wallis, W.A., 1952. Use of ranks in one-criterion variance analysis. J. Am. Stat. Assoc. 47 (260), 583–621. https://doi.org/10.1080/01621459.1952.10483441

Lester, B., Al-Rfou, R., Constant, N., 2021. The power of scale for parameter-efficient prompt tuning. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7–11 November, 2021. Association for Computational Linguistics, pp. 3045–3059. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.243

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

Li, X.L., Liang, P., 2021. Prefix-tuning: optimizing continuous prompts for generation. In: Zong, C., Xia, F., Li, W., Navigli, R. (Eds.), Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1–6, 2021. Association for Computational Linguistics, pp. 4582–4597. https://doi.org/10.18653/V1/2021.ACL-LONG.353

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., Tang, J., 2023. Gpt understands, too. AI Open https://doi.org/https://doi.org/10.1016/j.aiopen.2023.08.012

Longpre, S., Wang, Y., DuBois, C., 2020. How effective is task-agnostic data augmentation for pretrained transformers? In: Cohn, T., He, Y., Liu, Y. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16–20 November 2020. Association for Computational Linguistics, pp. 4401–4411. https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.394

Luo, Z., Li, K., Lin, H., Tian, Y., Kankanhalli, M.S., Ma, J., 2025. Tree-of-evolution: tree-structured instruction evolution for code generation in large language models. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (Eds.), Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27, - August 1, 2025. Association for Computational Linguistics, pp. 297–316. https://aclanthology.org/2025.acl-long.14/.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B.P., Hermann, K., Welleck, S., Yazdanbakhsh, A., Clark, P., 2023. Self-refine: iterative refinement with self-feedback. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (Eds.), Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10–16, 2023. http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Maertens, R., Petegem, C.V., Strijbol, N., Baeyens, T., Jacobs, A.C., Dawyndt, P., Mesuere, B., 2022. Dolos: language-agnostic plagiarism detection in source code. J. Comput. Assist. Learn. 38 (4), 1046–1061. https://doi.org/10.1111/JCAL.12662

Mishra, S., Khashabi, D., Baral, C., Hajishirzi, H., 2022. Cross-task generalization via natural language crowdsourcing instructions. In: Muresan, S., Nakov, P., Villavicencio, A. (Eds.), Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22–27, 2022. Association for Computational Linguistics, pp. 3470–3487. https://doi.org/10.18653/V1/2022.ACL-LONG.244

Mitchell, M., 1998. An introduction to genetic algorithms. MIT Press.

Moradi, M., Samwald, M., 2021. Evaluating the robustness of neural language models to input perturbations. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP

2021, Virtual Event / Punta Cana, Dominican Republic, 7–11 November, 2021. Association for Computational Linguistics, pp. 1558–1570. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.117

Nazzal, M., Khalil, I., Khreishah, A., Phan, N., 2024a. 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. Association for Computing Machinery, New York, NY, USA, p. 2266-2280. https://doi.org/10.1145/3658644.3690298

Nazzal, M., Khalil, I., Khreishah, A., Phan, N., 2024b. Promsec: prompt optimization for secure generation of functional source code with large language models (LLMs). In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (Eds.), Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14–18, 2024. ACM, pp. 2266–2280. https://doi.org/10.1145/3658644.3690298

Nie, Y., Williams, A., Dinan, E., Bansal, M., Weston, J., Kiela, D., 2020. Adversarial NLI: a new benchmark for natural language understanding. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J.R. (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5–10, 2020. Association for Computational Linguistics, pp. 4885–4901. https://doi.org/10.18653/V1/2020.ACL-MAIN.441

Onakpojeruo, E.P., Uzun, B., David, L.R., Ozsahin, I., Ozsahin, D.U., 2024. Selection techniques in genetic algorithm. In: 17th International Conference on Development in eSystem Engineering, DeSE 2024, Khorfakkan, United Arab Emirates, November 6–8, 2024. IEEE, pp. 411–416. https://doi.org/10.1109/DESE63988.2024.10912015

van Oort, B., Cruz, L., Loni, B., van Deursen, A., 2022. “project smells” - experiences in analysing the software quality of ML projects with mllint. In: 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22–24, 2022. IEEE, pp. 211–220. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794115

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2022. Asleep at the keyboard? Assessing the security of github copilot’s code contributions. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22–26, 2022. IEEE, pp. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B., 2023. Examining zero-shot vulnerability repair with large language models. In: 2023 2023 IEEE Symposium on Security and Privacy (SP) (SP). IEEE Computer Society, Los Alamitos, CA, USA, pp. 1–18. https://doi.org/10.1109/SP46215.2023.00001

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

Pilehvar, M.T., Camacho-Collados, J., 2019. Wic: the word-in-context dataset for evaluating context-sensitive meaning representations. In: Burstein, J., Doran, C., Solorio, T. (Eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, pp. 1267–1273. https://doi.org/10.18653/V1/N19-1128

Prasad, A., Hase, P., Zhou, X., Bansal, M., 2023. GrIPS: gradient-free, edit-based instruction search for prompting large language models. In: Vlachos, A., Augenstein, I. (Eds.), Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2–6, 2023. Association for Computational Linguistics, pp. 3827–3846. https://aclanthology.org/2023.eacl-main.277.

Pryzant, R., Iter, D., Li, J., Lee, Y.T., Zhu, C., Zeng, M., 2023. Automatic prompt optimization with “gradient descent” and beam search. In: Bouamor, H., Pino, J., Bali, K. (Eds.), Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6–10, 2023. Association for Computational Linguistics, pp. 7957–7968. https://doi.org/10.18653/V1/2023.EMNLP-MAIN.494

PyCQA, 2025. Bandit documentation. https://bandit.readthedocs.io/en/latest/index.html.

Qian, J., Dong, L., Shen, Y., Wei, F., Chen, W., 2022. Controllable natural language generation with contrastive prefixes. In: Muresan, S., Nakov, P., Villavicencio, A. (Eds.), Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22–27, 2022. Association for Computational Linguistics, pp. 2912–2924. https://doi.org/10.18653/V1/2022.FINDINGS-ACL.229

Rabbi, M.F., Champa, A.I., Zibran, M.F., Islam, M.R., 2024. AI writes, we analyze: the chatgpt python code saga. In: Spinellis, D., Bacchelli, A., Constantinou, E. (Eds.), 21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15–16, 2024. ACM, pp. 177–181. https://doi.org/10.1145/3643991.3645076

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

Rauf, I., Petre, M., Tun, T., Lopez, T., Lunn, P., vander Linden, D., Towse, J.N., Sharp, H., Levine, M., Rashid, A., Nuseibeh, B., 2022. The case for adaptive security interventions. ACM Trans. Softw. Eng. Methodol. 31 (1), 9:1–9:52. https://doi.org/10.1145/3471930

Roemmele, M., Bejan, C.A., Gordon, A.S., 2011. Choice of plausible alternatives: an evaluation of commonsense causal reasoning. In: Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21–23, 2011. AAAI. http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2418.

Ruohonen, J., Hjerppe, K., Rindell, K., 2021. A large-scale security-oriented static analysis of python packages in pyPI. In: 18th International Conference on Privacy, Security and Trust, PST 2021, Auckland, New Zealand, December 13–15, 2021. IEEE, pp. 1–10. https://doi.org/10.1109/PST52912.2021.9647791

Sakaguchi, K., Bras, R.L., Bhagavatula, C., Choi, Y., 2021. Winogrande: an adversarial winograd schema challenge at scale. Commun. ACM 64 (9), 99–106. https://doi.org/10.1145/3474381

Siddiq, M.L., Majumder, S.H., Mim, M.R., Jajodia, S., Santos, J. C.S., 2022. An empirical study of code smells in transformer-based code generation techniques. In: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 71–82. https://doi.org/10.1109/SCAM55253.2022.00014

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

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

Sun, T., He, Z., Qian, H., Zhou, Y., Huang, X., Qiu, X., 2022a. Bbtv2: towards a gradient-free future with large language models. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (Eds.), Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022. Association for Computational Linguistics, pp. 3916–3930. https://doi.org/10.18653/V1/2022.EMNLP-MAIN.259

Sun, T., Shao, Y., Qian, H., Huang, X., Qiu, X., 2022b. Black-box tuning for language-model-as-a-service. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvári, C., Niu, G., Sabato, S. (Eds.), International Conference on Machine Learning, ICML 2022, 17–23 July 2022, Baltimore, Maryland, USA. PMLR, pp. 20841–20855. https://proceedings.mlr.press/v162/sun22e.html.

Thomas, J., Harden, A., 2008. Methods for the thematic synthesis of qualitative research in systematic reviews. BMC Med. Res. Methodol. 8.

Tony, C., DíazFerreyra, N.E., Mutas, M., Dhif, S., Scandariato, R., 2025a. Prompting techniques for secure code generation: a systematic investigation. ACM Trans. Softw. Eng. Methodol. Just Accepted. https://doi.org/10.1145/3722108

Tony, C., Iannone, E., Scandariato, R., 2025b. Retrieve, refine, or both? Using task-specific guidelines for secure python code generation. In: IEEE International Conference on Software Maintenance and Evolution (Just Accepted), ICSME 2025, Auckland, Ne Zealand, September 10, - 12, 2025. IEEE, pp. 160–170.

Tony, C., Mutas, M., Ferreyra, N. E.D., Scandariato, R., 2023. Llmseceval: a dataset of natural language prompts for security evaluations. In: 20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15–16, 2023. IEEE, pp. 588–592. https://doi.org/10.1109/MSR59073.2023.00084

Vu, T., Lester, B., Constant, N., Al-Rfou’, R., Cer, D., 2022. Spot: better frozen model adaptation through soft prompt transfer. In: Muresan, S., Nakov, P., Villavicencio, A. (Eds.), Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22–27, 2022. Association for Computational Linguistics, pp. 5039–5059. https://doi.org/10.18653/V1/2022.ACL-LONG.346

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., Lyu, M.R., 2022a. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In: Roychoudhury, A., Cadar, C., Kim, M. (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14–18, 2022. ACM, pp. 382–394. https://doi.org/10.1145/3540250.3549113

Wang, J., Wang, C., Luo, F., Tan, C., Qiu, M., Yang, F., Shi, Q., Huang, S., Gao, M., 2022b. Towards unified prompt tuning for few-shot text classification. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022. Association for Computational Linguistics, pp. 524–536. https://doi.org/10.18653/V1/2022.FINDINGS-EMNLP.37

Wang, Z., Panda, R., Karlinsky, L., Feris, R., Sun, H., Kim, Y., 2023. Multitask prompt tuning enables parameter-efficient transfer learning. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023. OpenReview.net. https://openreview.net/pdf?id=Nk2pDtuhTq.

Wang, Z.Z., Asai, A., Yu, X.V., Xu, F.F., Xie, Y., Neubig, G., Fried, D., 2025. CodeRAG-bench: can retrieval augment code generation? In: Chiruzzo, L., Ritter, A., Wang, L. (Eds.), Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29, - May 4, 2025. Association for Computational Linguistics, pp. 3199–3214. https://doi.org/10.18653/V1/2025.FINDINGS-NAACL.176

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C., 2023. A prompt pattern catalog to enhance prompt engineering with chatGPT. CoRR abs/2302.11382. https://doi.org/10.48550/arXiv.2302.11382

Xiao, Y., Watson, M., 2019. Guidance on conducting a systematic literature review. J. Plann. Educ. Res. 39, 93–112. https://doi.org/10.1177/0739456X17723971

Xu, H., Chen, Y., Du, Y., Shao, N., Wang, Y., Li, H., Yang, Z., 2022. GPS: genetic prompt search for efficient few-shot learning. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (Eds.), Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022. Association for Computational Linguistics, pp. 8162–8171. https://doi.org/10.18653/v1/2022.emnlp-main.559

Xu, Z., Shen, Y., Huang, L., 2023. Multiinstruct: improving multi-modal zero-shot learning via instruction tuning. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (Eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9–14, 2023. Association for Computational Linguistics, pp. 11445–11465. https://doi.org/10.18653/V1/2023.ACL-LONG.641

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q.V., Zhou, D., Chen, X., 2024. Large language models as optimizers. In: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024. OpenReview.net. https://openreview.net/forum?id=Bb4VGOWELI.

Ye, S., Sun, Z., Wang, G., Guo, L., Liang, Q., Li, Z., Liu, Y., 2025. Prompt alchemy: automatic prompt refinement for enhancing code generation. IEEE Trans. Softw. Eng. Accepted. https://arxiv.org/abs/2503.11085.

Yu, Z., Wu, Y., Zhang, N., Wang, C., Vorobeychik, Y., Xiao, C., 2023. CodeIPPrompt: intellectual property infringement assessment of code language models. In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (Eds.), International Conference on Machine Learning, ICML 2023, 23–29 July 2023, Honolulu, Hawaii, USA. PMLR, pp. 40373–40389. https://proceedings.mlr.press/v202/yu23g.html.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., Choi, Y., 2019. Hellaswag: can a machine really finish your sentence? In: Korhonen, A., Traum, D.R., Màrquez, L. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers. Association for Computational Linguistics, pp. 4791–4800. https://doi.org/10.18653/V1/P19-1472

Zhang, B., Du, T., Tong, J., Zhang, X., Chow, K., Cheng, S., Wang, X., Yin, J., 2024. Seccoder: towards generalizable and robust secure code generation. In: Al-Onaizan, Y., Bansal, M., Chen, Y. (Eds.), Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12–16, 2024. Association for Computational Linguistics, pp. 14557–14571. https://aclanthology.org/2024.emnlp-main.806.

Zhao, J., Wang, Z., Yang, F., 2023. Genetic prompt search via exploiting language model probabilities. In: Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China. ijcai.org, pp. 5296–5305. https://doi.org/10.24963/IJCAI.2023/588

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B.Y., Fu, J., Chen, W., Yue, X., 2024a. Opencodeinterpreter: integrating code generation with execution and refinement. In: Ku, L., Martins, A., Srikumar, V. (Eds.), Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and Virtual Meeting, August 11–16, 2024. Association for Computational Linguistics, pp. 12834–12859. https://doi.org/10.18653/V1/2024.FINDINGS-ACL.762

Zheng, Y., Tan, Z., Li, P., Liu, Y., 2024b. Black-box prompt tuning with subspace learning. IEEE/ACM Trans. Audio Speech Lang. Process. 32, 3002–3013. https://doi.org/10.1109/TASLP.2024.3407519

Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J., 2023. Large language models are human-level prompt engineers. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023. OpenReview.net. https://openreview.net/pdf?id=92gvk82DE-.

Zhuo, J., Zhang, S., Fang, X., Duan, H., Lin, D., Chen, K., 2024. ProSA: assessing and understanding the prompt sensitivity of LLMs. In: Al-Onaizan, Y., Bansal, M., Chen, Y. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12–16, 2024. Association for Computational Linguistics, pp. 1950–1976. https://aclanthology.org/2024.findings-emnlp.108.