Prompting Techniques for Secure Code Generation: A Systematic Investigation

CATHERINE TONY, NICOLÁS E. DÍAZ FERREYRA, MARKUS MUTAS, SALEM DHIF, and RICCARDO SCANDARIATO, Hamburg University of Technology, Hamburg, Germany

Large Language Models (LLMs) are gaining momentum in software development with prompt-driven programming enabling developers to create code from Natural Language (NL) instructions. However, studies have questioned their ability to produce secure code and, thereby, the quality of prompt-generated software. Alongside, various prompting techniques that carefully tailor prompts have emerged to elicit optimal responses from LLMs. Still, the interplay between such prompting strategies and secure code generation remains under-explored and calls for further investigations. *Objective*: In this study, we investigate the impact of different prompting techniques on the security of code generated from NL instructions by LLMs. *Method*: First, we perform a systematic literature review to identify the existing prompting techniques that can be used for code generation tasks. A subset of these techniques are evaluated on GPT-3, GPT-3.5, and GPT-4 models for secure code generation. For this, we used an existing dataset consisting of 150 NL security-relevant code generation prompts. *Results*: Our work (i) classifies potential prompting techniques for code generation (ii) adapts and evaluates a subset of the identified techniques for secure code generation tasks, and (iii) observes a reduction in security weaknesses across the tested LLMs, especially after using an existing technique called Recursive Criticism and Improvement (RCI), contributing valuable insights to the ongoing discourse on LLM-generated code security.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Human-centered computing → Empirical studies in HCI**;

Additional Key Words and Phrases: LLMs, secure code generation, prompt engineering

ACM Reference format:
Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhif, and Riccardo Scandariato. 2025. Prompting Techniques for Secure Code Generation: A Systematic Investigation. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 225 (October 2025), 53 pages.
https://doi.org/10.1145/3722108

This work was partially supported by the EU-funded project Sec4AI4Sec: Cybersecurity for AI-Augmented Systems (grant no. 101120393).

Authors' Contact Information: Catherine Tony (corresponding author), Hamburg University of Technology, Hamburg, Germany; e-mail: catherine.tony@tuhh.de; Nicolás E. Díaz Ferreyra, Hamburg University of Technology, Hamburg, Germany; e-mail: nicolas.diaz-ferreyra@tuhh.de; Markus Mutas, Hamburg University of Technology, Hamburg, Germany; e-mail:markusmutas@googlemail.com; Salem Dhif, Hamburg University of Technology, Hamburg, Germany; e-mail: salem.dhif@tuhh.de; Riccardo Scandariato, Hamburg University of Technology, Hamburg, Germany; e-mail: riccardo.scandariato@tuhh.de.

ACM 1557-7392/2025/10-ART225
https://doi.org/10.1145/3722108

1 Introduction

Large Language Models (LLMs) have received major attention recently due to their high performance in solving **Natural Language (NL)** processing tasks. Alongside, their application to program synthesis has advanced significantly, allowing software developers to generate code from NL descriptions or prompts. Overall, this is achieved through vast training sets of code and documentation text extracted from open-source repositories. While this approach helps LLMs produce functional implementations, it offers no guarantees of correctness or quality, as it treats code simply as text, ignoring essential semantic information [44]. Moreover, open-source projects are known for containing security flaws [36, 103, 115, 116], making LLM-generated code prone to security vulnerabilities [76, 77].

Recent investigations [107] show that developers are gradually showing a preference for AI-driven code assistants to initiate their coding process. These tools offer a valuable starting point, aiding in the development process and alleviating the need to search for information online. However, when utilizing such AI assistants powered by LLMs, developers often display an over-reliance behavior that involves optimistic assumptions regarding the correctness and security of the generated code without thorough questioning [78, 94]. Findings from a user study conducted by Perry et al. [78] revealed that participants who had access to an AI assistant tended to produce insecure solutions more frequently compared to those who did not have access to such assistance. This emphasizes the importance of exploring avenues to strengthen the security incorporated by the LLMs in the code generated by them.

Motivation. Prompt engineering, the process of refining prompts to optimize the quality of responses generated by LLMs, has garnered significant attention following the emergence of LLMs like ChatGPT, BARD, and others. A variety of sophisticated prompting techniques have been developed for tasks such as text generation, classification, and problem-solving. Many of these techniques can be used by the end users to directly prompt or interact with LLM-powered tools and chatbots. Despite the abundance of research in this field, the correlation between such prompting strategies and secure code generation has not been thoroughly examined or documented in the existing literature. Specifically, the extent to which such techniques can guide LLMs toward producing secure implementations remains an open question. While models like GPT-3 continually advance, with each version improving upon its predecessor, the implications of these enhancements for security are unclear. This underscores the importance of investigating NL prompting techniques that have the potential to enhance the security of the code generated by LLMs.

In this work, we perform a literature review to identify potential prompting techniques that can be used for code generation followed by an in-depth analysis of the impact of these techniques on improving the security in LLM-generated code. For this, we elaborate on the following **Research Questions (RQs)**:

RQ1: What are the existing prompting techniques that can be used for code generation? To answer this, we performed a systematic literature review of papers that introduced different prompting techniques that can be potentially used for code generation.

RQ2: What is the impact of different prompting techniques on the security of LLM-generated code? For this, we conducted an in-depth analysis using a subset of prompting techniques identified in the literature review. A dataset called *LLMSecEval* [104], containing 150 NL prompts specifying coding tasks that could potentially lead to insecure code implementations, was used for our experiments. Experiments were conducted utilizing GPT-3, GPT-3.5, and GPT-4 models, due to their widespread usage and advanced NL processing and coding capabilities, which are crucial for exploring various prompting techniques. We mainly evaluated Python programs generated by the LLMs since it is

one of the most popular choice of languages for developers.[1] Additionally, to further explore the generalizability of the findings obtained from Python to other programming languages, we also examined C code generated by GPT-4. The code generated by the LLMs for the selected techniques was evaluated for security weaknesses using Bandit [82] as the primary tool and CodeQL [43] serving as a supplementary analysis.

Our findings reaffirm the fact that LLM-generated code contains a large number of security weaknesses mainly related to CWE-78, CWE-259, CWE-94, and CWE-330. We observed that integrating different prompting techniques has a positive impact on the security of code generated by LLMs, particularly noticeable in advanced models like GPT-4. Notably, a technique known as **Recursive Criticism and Improvement (RCI)** [50] has exhibited significant potential in mitigating security weaknesses in the generated code. Furthermore, we have observed distinct changes in the coding behavior of the models when security specifications are introduced to the prompts, offering insights that can be utilized to refine prompting techniques for secure code generation.

Contributions. This work makes the following contributions to the field of secure code generation using LLMs:

- To the best of our knowledge, we present the first *systematic inventory of prompting techniques* that are suitable for code generation. Often, papers in this field make an arbitrary selection of a few techniques, e.g., based on convenience or because other referenced papers do the same. This article highlights that a rich selection of techniques exists and incentivizes the community to explore the alternatives in their work.
- To simplify this exploration, we have translated a selection of these generic prompting techniques into seven actionable templates (see Table 2) that can be reused by the community as is, or with some adaptations for (secure) code generation. This effort is expected to stimulate the use of the different prompting techniques, beyond the usual suspects.
- We provide insights (and rankings) concerning the prompting techniques that are more promising for secure code generation with a focus on Python, but also exploring C. Interestingly, to the extent of our knowledge, the most promising technique for both languages has not been used in the related work for secure code generation (cf. the first point).

The rest of the article is organized as follows: Section 2 presents the existing work on using LLMs for (secure) code generation. Sections 3 and 4 present the approach used for the systematic literature review and the findings obtained from it. Following this, Sections 5 and 6 delve into the specifics of the security evaluation of code generated by LLMs using various prompting techniques and the results. Insights obtained from the results are elaborated in Section 7 followed by a discussion on the impact of data leakage on the results in Section 8. Section 9 addresses the limitations, while Section 10 brings the work to a close.

2 Related Work

This section presents prior research that delves into the use of LLMs for code generation and explores studies that assess the security aspects of code generated by LLMs.

2.1 Code Generation Using LLMs

There are several works (both published and unpublished) that evaluate the code generation capabilities of LLMs. The following are a few notable ones that are peer-reviewed.

[1] https://statisticstimes.com/tech/top-computer-languages.php.

A paper by Hendrycks et al. [38] evaluated the code generated by GPT-2 [83], GPT-3 [13], and GPT-Neo using a benchmark dataset called Automated Program Progress Standard [38] that consists of 10,000 NL coding problems along with corresponding test cases and ground truth solutions created by humans. For the evaluation, they employed the few-shot prompting technique where the model is provided with a set of <input-output> examples to demonstrate how to solve the problem. At the time of this study, they observed that the overall performance exhibited by the models was low based on the percentage of test cases passed. In another study conducted by Austin et al. [5], the authors explored the limitations of program synthesis carried out by language models trained at various scales, ranging from 244M to 137B parameters. To accomplish this, they created two datasets: the **Mostly Basic Programming Problems (MBPP)** dataset and the MathQA-Python dataset. The MBPP dataset comprises problem statements, simple Python functions designed to solve these problems, and three corresponding test cases. On the other hand, the MathQA-Python dataset presents mathematical problems, multiple-choice answers for these problems, and Python implementations that produce the correct answers. Both datasets are created to verify the semantic correctness of the generated Python programs. They also employed a few-shot prompting technique and their observations revealed a correlation between the increase in model size and improved performance.

Xu et al. [119] conducted a comprehensive assessment of various LLMs, including Codex [16], GPT-J, GPT-Neo, GPT-NeoX-20B [12], CodeParrot [59], and PolyCoder (a model developed by the authors of this article) for their code generation capabilities. Their evaluation focused on these models' performance using the HumanEval [16] dataset, which contains 164 distinct coding tasks presented as prompts with corresponding test cases. These prompts consist of incomplete code snippets paired with NL comments rather than a complete NL instruction describing the task. In this study, they employed a zero-shot prompting technique. Zero-shot prompting entails not providing explicit <input-output> pairs to the LLMs to demonstrate how to approach the given task. Based on this study, Codex emerged as the top-performing model, outperforming all the other models in the evaluation.

A study by Zeng et al. [127] tried to understand how pre-trained models perform for program understanding and generation tasks by experimenting with eight LLMs that include CodeBERT [28], GraphCodeBERT [33], ContraCode [45], CodeGPT, PLBART [2], CodeTrans [26], CoText [79], and CodeT5 [111] mainly using the CodeXGLUE [64] benchmark. This benchmark is a collection of datasets spread across 10 different code-related tasks. The dataset used for code generation tasks within this benchmark is known as Concode. The prompts in Concode encompass NL problem descriptions, structured in the form of Java Doc comments and class environments. The researchers employed zero-shot prompting to evaluate the models. The results of their experiments indicated that CodeT5 and CodeTrans consistently delivered the highest performance in code generation tasks. In another work, an extensive literature review was conducted by Fan et al. [27] where they examine papers that present works done using LLMs for software engineering tasks. Their analysis reveals a growing emphasis on models from the GPT series, with GPT-4 [73] gaining significant attention in studies related to code generation using LLMs.

Besides the aforementioned studies, there exist papers introducing code synthesis benchmarks like EvalPlus [63] and Multipl-E [15], which assess the code generated by various LLMs. Furthermore, the papers that introduce different LLMs capable of performing code generation [16, 18, 28, 29, 58, 111, 119] task also perform evaluation of the code generated by their respective models. The prompting techniques employed in such studies are predominantly limited to either zero-shot or few-shot prompting.

Motivation 1. Despite the extensive research in the domain of code generation by LLMs, there is a lack of papers that explore various prompting techniques other than *zero-shot* and *few-shot* prompting to enhance the code generation capabilities of LLMs.

2.2 Security in LLM-Generated Code

As mentioned earlier, prior work has elaborated on the security of code generated by LLMs. Pearce et al. [76], for instance, used 54 high-risk security scenarios containing incomplete code snippets (C and Python) to assess code completions produced by GitHub Copilot and observed that 40% of them contained security vulnerabilities. However, a study by Asare et al. [4] compared C/C++ code generated by human developers against the ones generated by Copilot and observed that Copilot is not as bad as humans in introducing vulnerabilities in code. The experiments in these studies were done using zero-shot prompts. In another work by Pearce et al. [77], they tested the code repair capabilities of LLMs using various program repair scenarios. Overall, they concluded that Codex and Jurassic-1 [60] are capable of finding fixes for simple scenarios again under zero-shot settings. Jesse et al. [46] did a recent study where they examined if Codex and other LLMs generate **Simple, Stupid Bugs (SStuBs)** and found that these models produce twice as many SStuBs as correct code. On the other hand, [37] proposed a learning approach for controlled code generation called SVEN. Such an approach, in which a Boolean parameter is passed to enforce secure/insecure code generation, increased the number of secure code produced by an LLM called CodeGen by 25%. Another study by Yetiştiren et al. [124] assessed the quality (i.e., validity, correctness, reliability, security, and maintainability) of code generated by Copilot, Amazon CodeWhisperer, and ChatGPT using the *HumanEval* dataset. Notably, no significant security vulnerabilities were found in the generated code. However, the authors acknowledge the limitations of their security evaluation, since the HumanEval dataset is designed to verify functional correctness rather than code security.

Delving further into the realm of secure code generation using LLMs, Sandoval et al. [93] investigated the impact of LLM on code security through a user study. The study involved 58 computer science students who were tasked with performing simple operations on a linked list using C programming language with a focus on memory-based vulnerabilities. They observed that the participants who used an AI assistant powered by Codex introduced security-critical bugs at a rate no higher than 10% when compared to the control group indicating that the use of LLMs does not introduce new vulnerabilities. Nevertheless, it is essential to acknowledge that these findings may not be universally applicable to more complex programming tasks. Contrary to the previous study Perry et al. [78] observed different results in a study that explored developers' interactions with AI code assistants concerning security. Forty-seven participants were engaged with an AI assistant powered by Codex to fulfill five security-related programming tasks across Python, JavaScript, and C. The findings revealed that participants utilizing AI assistants were prone to generating insecure solutions more often than those without AI assistance in four out of five tasks. Typical issues encompassed the selection of unsafe libraries, incorrect library utilization, insufficient comprehension of edge cases involving external entities like file systems or databases, and inadequate sanitization of user input.

Additionally, apart from empirical and user studies on LLMs, a systematic literature review conducted by Yao et al. [122] delves into the use of LLMs for security and privacy purposes. Their findings indicate a plethora of works employing LLMs in security-related tasks, such as coding, test case generation, bug detection, vulnerability detection, and fixing. These endeavors have positively influenced research within the security community. However, none of these studies thoroughly investigate different prompting techniques to enhance the secure code generation process.

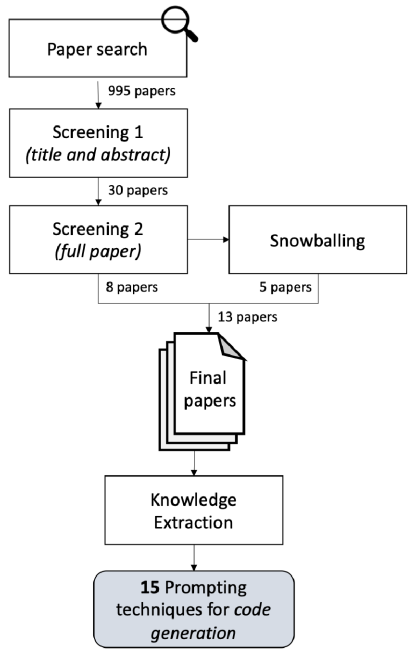

Fig. 1. Steps followed for the SLR on prompting techniques that are suitable for code generation.

Motivation 2: Studies we have seen so far do not thoroughly explore the impact of prompting techniques to improve the security of the code generated by the LLMs. This underscores the need for further research to identify such techniques that can improve the secure code generation capabilities of LLMs.

3 Methodology for Systematic Literature Review

The goal of this review is to find prompting techniques that can be used for code generation tasks using LLMs. However, there are only a limited number of prompting techniques explicitly designed for code generation. Therefore, we opted to review all prompting techniques introduced for generating textual content, presuming their potential transferability to code generation tasks, given that code generation falls within the domain of textual content generation. The steps followed to perform the literature review are depicted in Figure 1. We used the *Publish or Perish* tool [35] to retrieve papers from Google Scholar. Following the PICOC strategy [14], the search query given below was employed to retrieve the relevant papers that introduce prompting techniques for textual content generation.

```
prompt* AND (engineer* OR pattern* OR technique*) AND (language model* OR pre-
trained model* OR llm* OR ptm*)
```

The search was conducted in October 2023. The results of this search were examined in their ranked order following the steps described below.

Paper Screening. The review process was done in two screening steps. In the *first screening*, we looked at the title and abstract of the paper to decide if it was relevant to our study. If it is then it was shortlisted for the *second screening*. In the second screening, we looked into the full paper

to decide if it fits our criteria. The first and second screening was done based on the following inclusion and exclusion criteria.
Inclusion Criteria:

IC1: Paper deals with prompting LLMs using one or more techniques
IC2: Paper is published since 2018:
IC3: Paper is written in English

Exclusion Criteria:

EC1: Paper does not introduce new prompting techniques to query LLM
EC2: Paper deals with the generation of anything other than text and code (e.g., image, speech, and video data)
EC3: Paper that presents prompting techniques that cannot be used for generation tasks (e.g., techniques specific to classification tasks)
EC4: Paper that presents automated prompt optimization techniques and frameworks (e.g., prompt tuning and black-box tuning)
EC5: Paper that presents prompting technique for attacking the model (e.g., jailbreak prompts)
EC6: Out of scope (e.g., techniques for medical science)

IC1 is the main criteria that we use to include papers in the review since our goal is to find papers that explore different ways to prompt LLMs to optimize the response. Significant developments in the field of LLMs started happening since the year 2018 (GPT-1,[2] BERT [21]). Hence we defined *IC2* to look at relevant works on prompting techniques that emerged after this. *IC3* is a basic criterion that only includes papers written in English.

The exclusion criteria are designed to identify prompting techniques that can be used for code generation even though they are not specifically created for this task. If any of the criteria outlined in EC are met, regardless of the IC, then a publication is disqualified from the review process. There are several works that use LLMs for different tasks through prompting. However, many of these works adhere to basic prompting methods without introducing any novel techniques. As our objective is to identify and list novel prompting approaches, we employed *EC1* as the primary criterion for filtering out papers that rely on existing techniques. *EC2* excludes papers focusing on generating anything other than textual content, and by extension code. This decision is based on the differing training methodologies between LLMs handling non-textual data such as videos or images and those dealing with textual data. Consequently, we proceeded with the assumption that prompting techniques for non-textual data may not be suitable for code generation. *EC3* eliminates techniques that target problems with restrictive answers, such as yes/no questions, cloze-style questions, or multiple-choice questions. These techniques are excluded because they do not facilitate generation tasks like code generation. Automated prompt engineering techniques such as prompt tuning [109] and black-box tuning [34, 99] as well as automated frameworks that optimize prompts and LLM outputs [121, 132] are excluded from our list as they follow a very different methodology involving data training, learning, external tools or complex automated algorithms to improve prompts. Evaluating such techniques requires a different setup compared to non-automated prompting methods. Consequently, papers presenting these techniques are removed using *EC4*. Additionally, papers presenting various prompts and techniques aimed at attacking a model are excluded using *EC5*, as they are not suitable for code generation. Papers discussing topics outside of prompt engineering or belonging to irrelevant fields, such as medicine or construction, are eliminated using *EC6*.

[2] https://openai.com/research/language-unsupervised.

We reached saturation at the mark of 358 search results, as we observed no new papers that passed the first screening process within over 100 results before that point. Consequently, we concluded this stage upon reaching the 358th paper in the ranked results obtained from our search. Following the first screening of titles and abstracts, 30 of them were chosen to undergo further evaluation. Out of the 358 papers, the majority were excluded based on *EC1*, which involves eliminating works that do not introduce a new prompting technique. Upon full review in the second screening step, 22 papers were excluded, leaving a selection of eight relevant papers introducing novel prompting techniques.

Snowballing. To ensure that we did not miss any other relevant papers, we also performed three rounds of backward snowballing [117]. Here we went through the references of the selected papers iteratively following the same two-step screening process as above until no new papers were obtained. From this, we obtained five additional relevant ones making the total number of relevant papers 13. Three papers under consideration were released on preprint servers like arXiv and have not undergone formal peer review. However, these preprint papers have been frequently cited with the least number of citations being 48. Hence we decided to retain those papers.

Knowledge Extraction. Each final paper that introduced a prompting technique suitable for code generation was examined in detail. The primary objective was to extract the techniques themselves and pinpoint their key features. For this, we performed a lightweight thematic analysis with open coding as it offers a qualitative method for analyzing textual or qualitative data to interpret patterns or themes within the data [102, 118]. During this process, the first author extracted codes related to prompting techniques following an inductive approach. The themes that emerged from this coding were then discussed with two other authors to categorize and label the techniques.

In addition to this, attention was also directed toward details such as the LLMs on which the technique was tested, the specific tasks used for evaluation, and the datasets employed for this purpose. Furthermore, data regarding the year of publication, venue, and citation count at the time of the study were also extracted. This was aimed at creating a consolidated source of information beneficial to researchers and developers delving into prompting techniques for code generation.

4 Prompting Techniques for Code Generation (*RQ1*)

In this section, we present an overview of the selected prompting techniques from the SLR that are deemed suitable for code generation tasks. Throughout our review, we encountered numerous prompting techniques. However, not all of them were selected to be in our final list as determined by our exclusion criteria. All results of this literature review, along with the techniques that were excluded from our consideration and the reason for their exclusion are documented in our replication package specified in Section 11.

4.1 Overview of the Selected Papers

The information extracted from the 13 papers is presented in Table 1. The chosen papers are those that introduce novel prompting techniques. Among these, we identified 15 distinct techniques designed for textual content generation with potential applicability to code generation tasks. Ten of these papers have undergone peer review, while the remaining have received at least 48 citations. Except for two papers [88, 113], all have conducted experimental validation of their introduced prompting techniques. Only two of them [47, 66] have evaluated their techniques specifically for code generation tasks. The other techniques primarily target various reasoning tasks such as symbolic, logical, commonsense, and arithmetic. Among the papers that conducted experimental validation, 10 out of 11 utilize OpenAI models indicating the prevalence of these models in research in this field.

Table 1. Final List of Prompting Techniques Obtained from the SLR That Can Potentially Be Used for Code Generation

Work	Prompting Techniques				Scope			Publication Details		
	Name	Single/ Multi-Step	Demonstrative/ Non-Demonstrative	Linear/ Parallel	Evaluation Task(s)	LLM(s)	Dataset(s)	Year	Venue	Citations
Brown et al. [13]	Zero-shot	S	N	L	Language modeling, question answering, translation, commonsense, reading comprehension, reasoning, inference, and arithmetic	GPT-3	PTB [68], LAMBADA [74], StoryCloze [71], HellaSwag [126], Natural Questions [54], WebQuestions [9], TriviaQA [48], WMT [25], WinoGrande [92], PIQA [11], ARC [10], UnifiedQA [49], OpenBookQA [70], CoQA [87], QuAC [17], DROP [23], SQuAD [85], RACE [55], SUPERGLUE [108], RTE [13], ANLI [72], SAT analogy [106]	2020	NeurIPS	24786
	One-shot	S	D	L						
	Few-shot	S	D	L						
Reynolds et al. [88]	Memetic Proxy	S	N	L	N/A	N/A	N/A	2021	CHI	544
Kojima et al. [51]	Zero-shot CoT	M	N	L	Arithmetic, symbolic, and logical reasoning	InstructGPT, PaLM	SingleEq [52], AddSub [40], MultiArith [90], AQUARAT [62], GSM8K [19], SVAMP [75], Last Letter Concatenation [112], Coin Flip [112], CommonsenseQA [101], StrategyQA [32], BIG-bench effort [98]	2022	NeurIPS	1901
Lampinen et al. [57]	Few-shot explanation	S	D	L	Reasoning, inference	Decoder-only Transformers (1B to 280B parameters)	BigBench Effort [98]	2022	EMNLP Findings	197
Wang et al. [110]	Self-consistency	M	D	P	Arithmetic and commonsense reasoning	UL2, GPT-3, LaMDA, PaLM	GSM8K [19], SVAMP [75], AQuA [62], StrategyQA [32], and ARC-challenge	2022	ICLR	626

(Continued)

Table 1. Continued

Work	Prompting Techniques				Scope			Publication Details		
	Name	Single/ Multi-Step	Demonstrative/ Non-Demonstrative	Linear/ Parallel	Evaluation Task(s)	LLM(s)	Dataset(s)	Year	Venue	Citations
Wei et al. [112]	CoT	M	D	L	Symbolic and commonsense reasoning	GPT-3, LaMDA, PaLM	CommonSenseQA [101], StrategyQA [32], BigBench effort [98], SayCan [42], Last letter concatenation [112], Coin flip [112]	2022	NeurIPS	4584
Zhou et al. [131]	Least-to-most	M	D	L	Symbolic manipulation, compositional generalization, math reasoning	GPT-3	Last Letter Concatenation [112], SCAN [56], GSM8K [19], DROP [23]	2022	ICLR	672
Fu et al. [30]	Complexity-based	M	D	P	Arithmetic, commonsense, temporal, and referential reasoning	LaMDA, PaLM Minerva, GPT-3 Codex, DiVeRSe	GSM8K [19], StrategyQA [32], MathQA [5], MultiArith [90], Penguin [100] Date Understanding [100]	2023	ICLR	194
Jiang et al. [47]	Self-planning	M	D	L	Code generation and completion	Codex	MBPP-sanitized [5], MBPP-ET [22], HumanEval [16], HumanEval-X [130], and HumanEval-ET [22]	2023	arXiv	48
Kim et al. [50]	Recursive ciriticism and improvement	M	N	L	Arithmetic and commonsense reasoning	InstructGPT3 + RLHF	SingleEq [52], AddSub [40], MultiArith [90], AQuA [62], GSM8K [19], SVAMP [75], CommonSenseQA [101], and StrategyQA [32]	2023	NeurIPS	135
Madaan et al. [66]	Self-refine	M	D	L	Sentiment reversal, dialog response, code optimization, code readability, math reasoning, acronym generation, constrained generation	GPT-3.5, GPT-4, Codex	Yelp reviews [128], FED [69], PIE [65], CodeNet [81], GSM8K [19], Acronyms, CommonGen [61]	2023	NeurIPS	430
White et al. [113]	Persona	S	N	L	N/A	N/A	N/A	2023	arXiv	580
Zheng et al. [129]	Progressive-hint	M	N	L	Arithmetic, reasoning	GPT-3, GPT-3.5-Turbo, GPT-4	AddSub [40], MultiArith [90], SingleEQ [52], SVAMP [77], GSM8K [19], AQuA [62], MATH [39]	2023	arXiv	84

Based on commonalities derived from the thematic analysis, we have labeled the techniques using three distinct properties related to their execution as shown in Table 1. They are *S*ingle/*M*ulti-step, *D*emonstrative/*N*on-*D*emonstrative, and *L*inear/*P*arallel. A technique that prompts the model in a single step, obtaining the final output with just one prompt, is referred to as a *single-step* technique. Conversely, a technique requiring multiple prompts to generate the final output is termed a *multi-step* technique. Single-step techniques are cost-effective compared to multi-step techniques as they necessitate only one prompt. Among the 15 techniques identified, 6 are single-step techniques, while the rest are multi-step techniques. If a technique is executed by providing demonstrative examples of inputs and expected outputs for prompting the model, it is categorized as a *demonstrative* technique. Conversely, a technique not requiring input-output examples is labeled as a *non-demonstrative* technique. Six out of 15 techniques are non-demonstrative.

Although demonstrative techniques may potentially yield desired outputs more effectively than non-demonstrative techniques, this depends on the availability of high-quality demonstrative examples. In real-world scenarios, especially in complex code generation tasks, obtaining such examples can be challenging. Most techniques in our inventory involve conducting a single sequential interaction with the LLM. Here, the model is prompted, and its response is either used as the final output or serves as a basis for proceeding to the next step of prompting. These techniques are labeled as *linear*.

Conversely, techniques that engage in multiple parallel chains of conversation with the model and utilize the parallel responses generated by the model to either finalize the output or advance to the next parallel step of prompting are labeled as *parallel*. Among the 15 techniques examined, only two are classified as parallel. However, there are techniques outside of our list that employ parallel response generation or interactions, such as *Ask Me Anything* [3] and *Tree-of-Thoughts* [121] which were excluded due to their unsuitability for our use case. Therefore, we opted to maintain this label within our list of techniques.

A more detailed description of the individual techniques included in Table 1 and how they can be used for code generation are presented in the following subsection.

4.2 Classification of Prompting Techniques

Aside from the labels provided in Table 1 (*single/multi-step, demonstrative/non-demonstrative*, and *linear/parallel*), we also identified some other common characteristics based on the strategic design of different prompting techniques which we used to classify them into five different categories as shown in Figure 2. Below we describe the categories and the techniques that belong to them, accompanied by demonstrations of how these techniques can be utilized for code generation tasks. The responses of the LLM depicted in these demonstrations were generated by ChatGPT (GPT-3.5), which is a conversational chatbot, in response to different prompting techniques.

4.2.1 Root Techniques. These are the foundational and most popular techniques based on which more advanced techniques are built. *Zero-shot, one-shot,* and *few-shot* prompting come under this category.

Zero-Shot. In this technique a model is asked to perform a task without task-specific training or examples at the time of inference [13]. In such cases, the model relies completely on the data it has seen during its pre-training to generate an appropriate response. In conversational LLMs such as ChatGPT, zero-shot prompting is possibly the most commonly used way of interaction by an average user. It has the advantage of not having to prepare a task-specific dataset of input-output demonstrations to generate desirable output. However, if the model has not seen data related to the task at hand in its training, then the performance of the model can be suboptimal with zero-shot

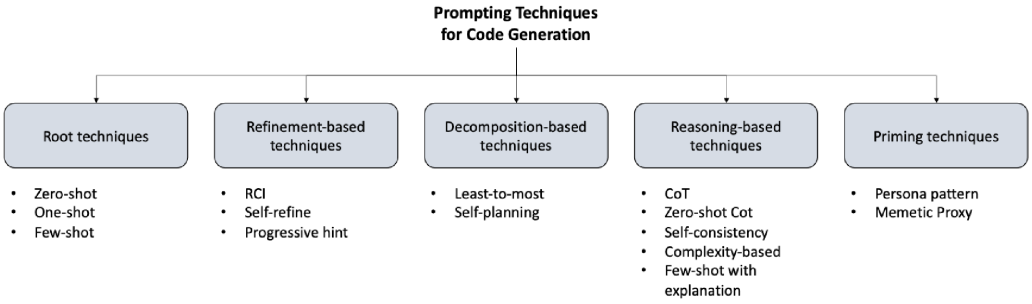

Fig. 2. Classification of prompting techniques for code generation.

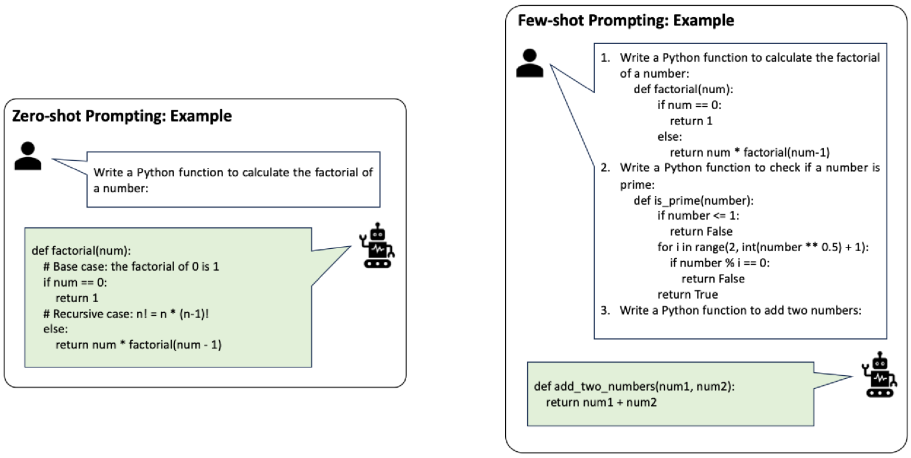

Fig. 3. Zero-shot *(left)* and few-shot *(right)* prompting with ChatGPT for code generation.

prompting. This technique can be directly used for code generation tasks. Figure 3 includes a demonstration of zero-shot prompting for a simple coding task and ChatGPT's response to it.

One-Shot/Few-Shot. One-shot and few-shot prompting techniques [13] are very similar to each other. In one-shot prompting, the model is given a single input-output example whereas in the few-shot prompting the model is given examples of the task at inference time as conditioning before providing the final input for which it is expected to produce the output. By supplying the model with both input and corresponding output samples, it gains the benefit of producing a response that closely aligns with the desired format. However, it can be a disadvantage when one does not have sufficient or relevant task-specific data in advance. An illustration demonstrating the application of few-shot prompting on ChatGPT for code generation can be seen in Figure 3. This example utilizes two few-shot examples. We have omitted a separate example of one-shot prompting since it closely resembles few-shot prompting but with only one demonstrative example.

4.2.2 Refinement-Based Techniques. Techniques belonging to this category focus on improving, refining, or iterating the model outputs. They might involve feedback loops, user interactions, or model self-assessment to enhance the quality of the generated responses. The prompting techniques that come under this category include RCI, self-refine, and **Progressive Hint Prompting (PHP)**.

RCI. This prompting technique [50] is built on the understanding that LLMs possess a strong capability to evaluate and recognize flaws in their own output. This technique involves a two-step

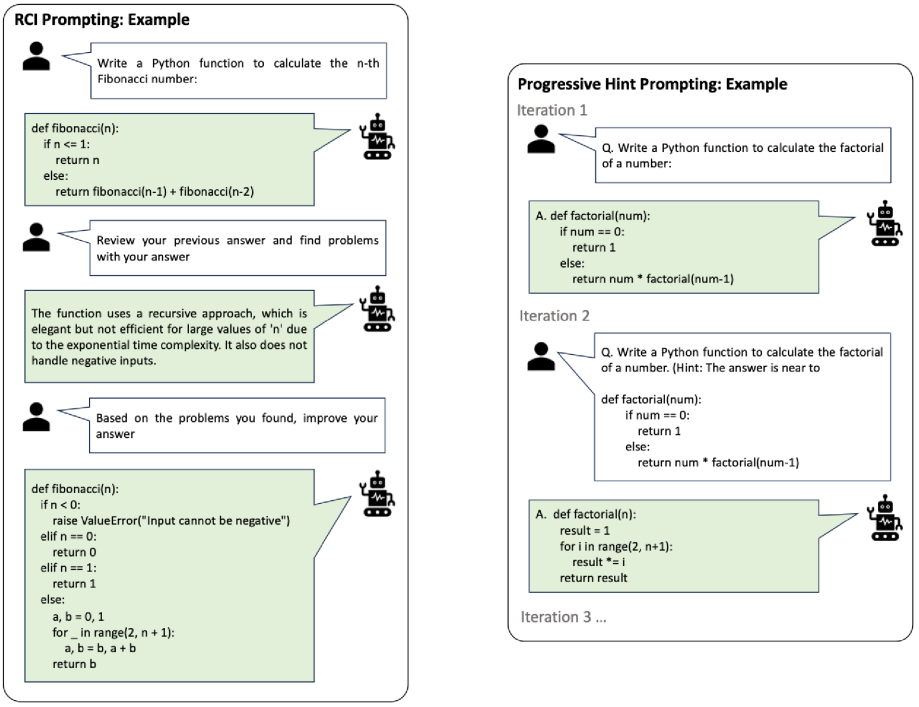

Fig. 4. RCI (left) and progressive hint *(right)* prompting with ChatGPT for code generation.

process in addition to providing the initial input task. First, the LLM is prompted to analyze and critique its current response (for instance: "Review your previous answer and find problems with your answer"). Subsequently, drawing from the critiques it has outlined, the LLM is then instructed to rectify the identified issues and revise its output accordingly (for example: "Based on the problems you found, improve your answer"). This two-step process is repeated until a satisfactory output is obtained or until a predefined number of iterations is done. RCI has the advantage that it needs no task-specific expert data to generate desirable responses. However, this approach can be expensive due to the iterative nature of the process. An added disadvantage is that the success of this approach relies on the ability of the model to identify its own mistakes. A demonstration of one iteration of this technique used for a code generation task is shown in Figure 4.

Self-Refine. This technique [66] is very similar to RCI. It uses two steps called *feedback* and *refine* in addition to an initial output generation step to generate high-quality output. The initial output from model M is generated using a task-specific prompt p_{gen} with few-shot <input, output> example pairs. Next, they use a prompt p_{fb} to generate feedback for the previously generated output by M. Few-shot examples are provided in this step in the form of <input, output, feedback> triplets. The next step is to refine the output based on the generated feedback. This is done using prompt p_{refine} that contains few-shot examples of refining outputs in the form of <input, output, feedback, refined> quadruples. An adaptation of this technique for code generation is shown in Figure 5.

Progressive Hint. PHP [129] is another technique that iteratively refines the output from the LLM by providing increasingly informative hints in each iteration. The pipeline of this approach is divided into two stages. The first stage is called *base answer and base prompt*. In this stage, the model is provided with an input task with a basic prompt to which a base answer is generated.

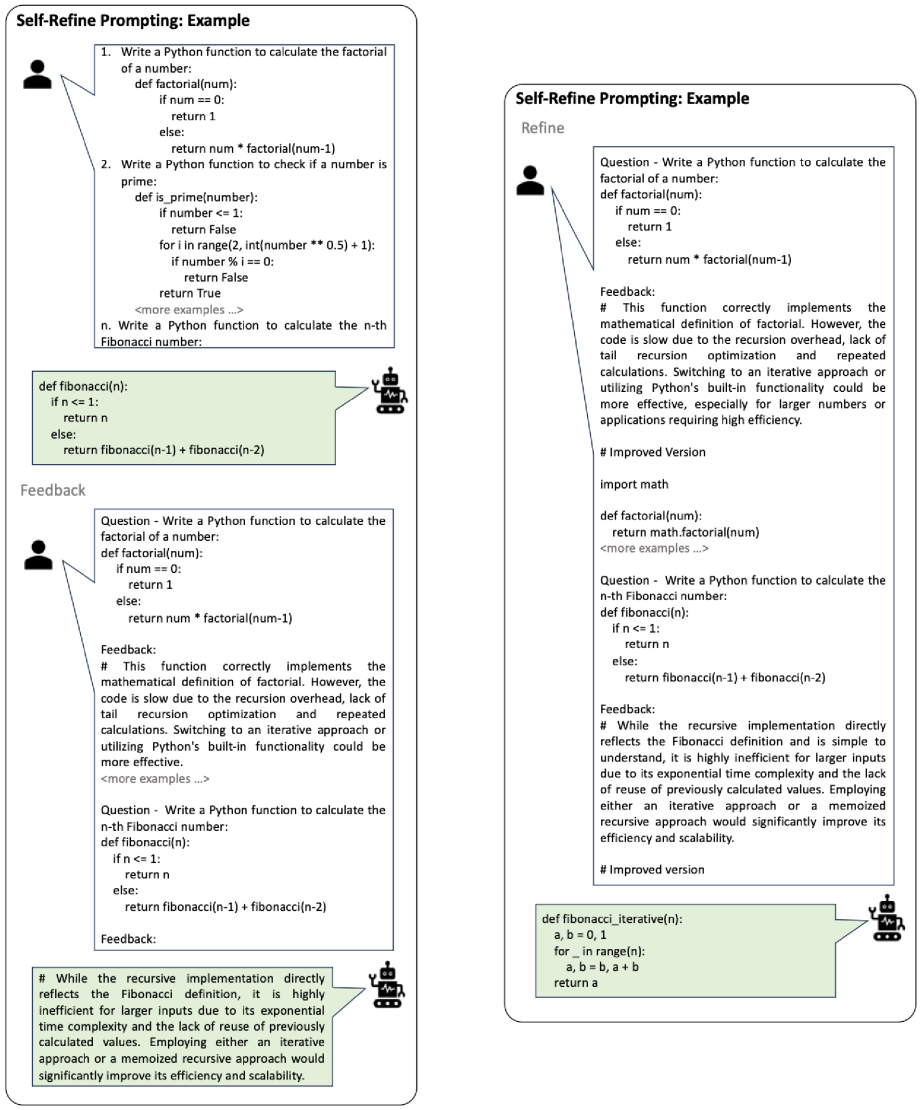

Fig. 5. Feedback generation *(left)* and refining *(right)* steps of self-refining prompting with ChatGPT for code generation.

The second stage is called *subsequent answer and PHP* where the base prompt is combined with hints that are extracted from the previous answers (or base answer in this case). This is repeated until the answers from the model do not change. Figure 4 shows the interaction with ChatGPT for a simple coding task using this technique. PHP can be combined with standard zero-shot prompting or sophisticated techniques such as **Chain-of-Thought (CoT)**. This approach requires at least two iterations. The approach is not considered successful until the last two outputs from the model are the same. This can become computationally expensive based on the task and the model. Additionally, the model can be misled if the hints provided stray too far away from the

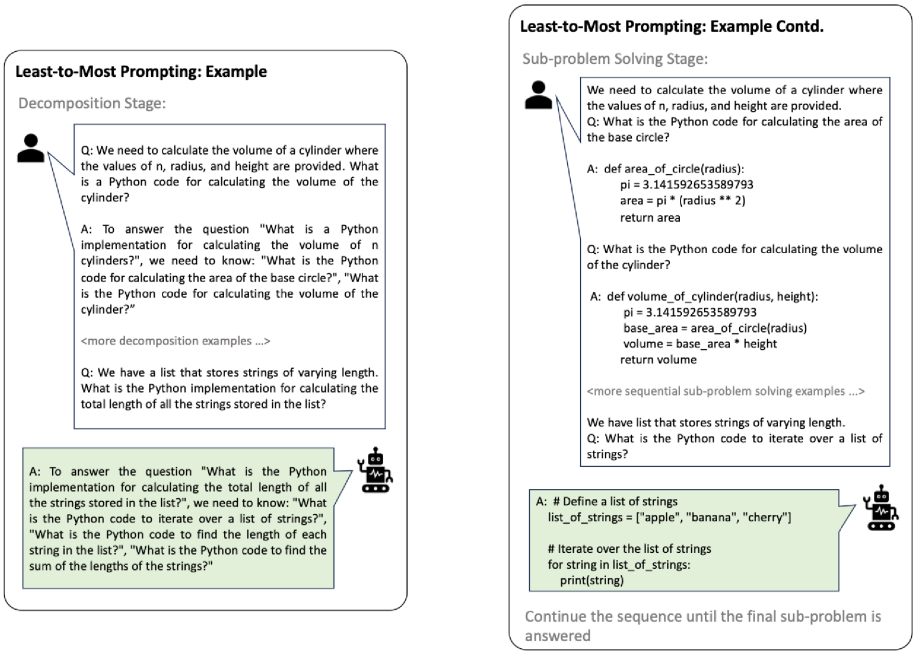

Fig. 6. Decomposition *(left)* and sub-problem solving *(right)* stage of least-to-most prompting with ChatGPT for code generation.

correct answer [129]. This approach can be theoretically used for code generation tasks as shown in Figure 4.

4.2.3 Decomposition-Based Techniques. Techniques in this category break down complex tasks or prompts into simpler, more manageable pieces. Here, the language models perform multiple small tasks to incrementally build toward the final, complex solution, facilitating more accurate responses. The techniques under this category include *least-to-most* and *self-planning* prompting.

Least-to-Most. This prompting technique [131] is executed in two stages. In the *decomposition* stage, the model is prompted to decompose the complex task into smaller sub-tasks. This prompt is delivered using a few-shot approach, where a few examples are presented to illustrate how larger tasks can be dissected into sub-tasks, followed by the actual complex task that needs to be addressed. The second stage is the *sub-problem solving* stage where the model is asked to sequentially solve all the sub-problems or sub-tasks identified in the decomposition stage. Here also, few-shot examples demonstrating how sub-problems are solved are provided. Responses derived from solving each sub-task are integrated back into the original task description before presenting the subsequent sub-task to the model. This iterative process continues until all sub-tasks have been resolved, resulting in the final solution.

Least-to-most prompting technique can also be used in combination with CoT or self-consistency prompting techniques. Similar to other advanced prompting methods, this technique's drawback is the necessity to supply few-shot examples for both the decomposition of a complex task and the resolution of its sub-tasks. The resource demands can escalate with the increasing number of sub-tasks involved in the process. This approach can also be potentially used for code generation provided you have a sufficient dataset containing information on coding problem decompositions and solutions. Figure 6 demonstrates how this technique can be used for code generation.

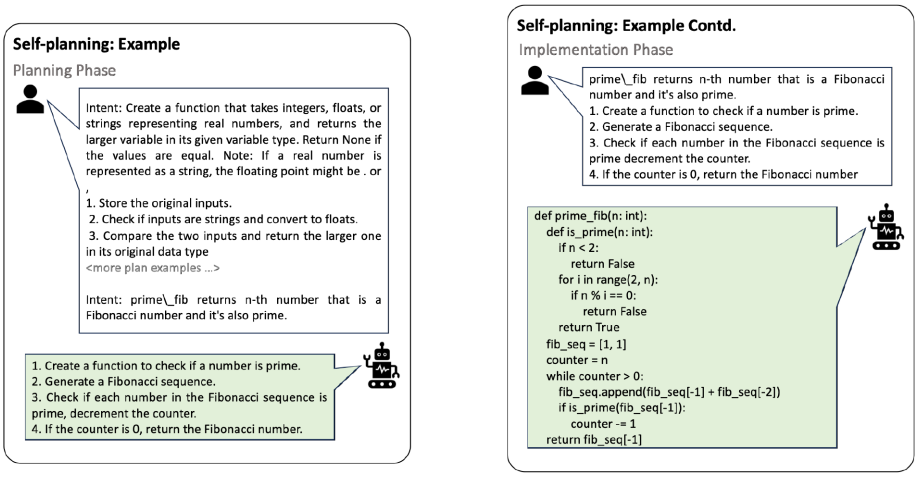

Fig. 7. Planning *(left)* and implementation *(right)* phase of self-planning prompting for code generation (borrowed from [47]).

Self-Planning. This prompting approach [47] is specifically designed for code generation problems. Hence no additional adaptation is required to tailor the technique for code generation tasks. Self-planning is carried out in two phases. The first one is the planning phase where the code generation task is decomposed into a plan of actions. This decomposition is done by the LLM itself. The LLM is provided with demonstrative examples of how to come up with plans to solve coding tasks before asking it to generate a plan for the task at hand. The action plan is structured as an ordered list of steps. The plan should always conclude with a return statement. The second phase is called the implementation phase wherein the LLM's formulated plan is integrated with the original task prompt. This integration prompts the LLM to adhere to its own outlined strategy when producing the final code snippet. An example demonstration of this prompting technique is shown in Figure 7. This example is directly taken from the original paper itself.

4.2.4 Reasoning-Based Techniques. Techniques that guide the model to employ and demonstrate logical reasoning for generating responses are categorized as reasoning-based techniques. Reasoning encompasses the act of drawing logical conclusions, evaluating arguments, and making inferences using the information at hand [41]. These methods emphasize the model's ability to engage in cognitive and logical processes. Rather than simplifying a task as in the case of decomposition-based techniques, these techniques encourage the model to follow a logical reasoning path and articulate its thought process. The techniques that come under this category are *CoT, zero-shot CoT, self-consistency,* and *few-shot with explanation.*

CoT. In this prompting approach [112], the LLM is compelled to produce a sequence of intermediary logical reasoning steps in NL, culminating in the solution to the presented problem. The goal of this approach is to replicate how humans solve a complex problem following a chain of reasoning or justification steps. In this method, the model is initially given a set of few-shot examples, consisting of <*input, CoT, output*> triplets, to guide its understanding before it tackles the actual task. This technique has been evaluated on various benchmarks including arithmetic, common sense, and symbolic reasoning. However, one can assume that CoT can also be applied to code generation tasks. Figure 8 demonstrates the CoT prompting technique for code generation.

An approach similar to this was proposed in 2017 by Ling et al. [63] where they train an attention-based sequence-to-sequence model to solve complex mathematical problems using a

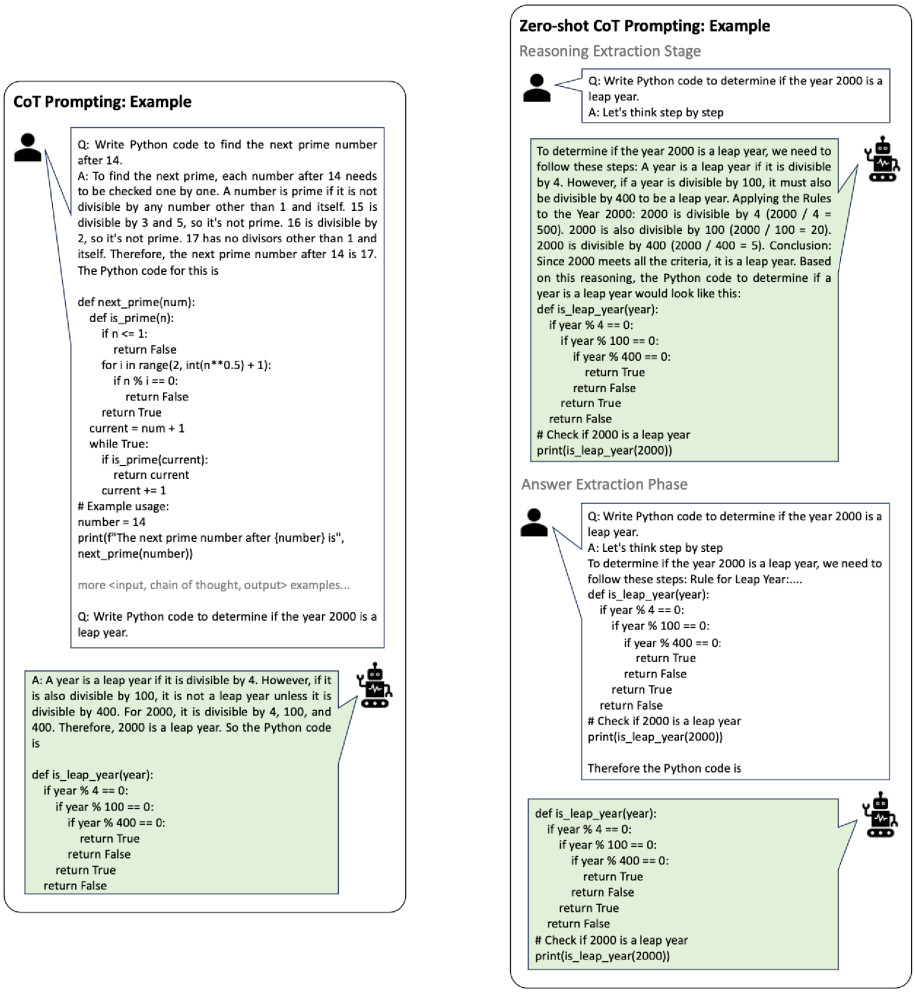

Fig. 8. CoT (left) and zero-shot CoT *(right)* prompting using ChatGPT for code generation.

dataset containing problems with answer rationales and the final correct answers. However, this approach focused on training rather than explicitly prompting a model, and it did not involve an LLM. Hence we identify CoT as a novel prompting technique.

Zero-Shot CoT. This approach [51] addresses the limitations of the CoT approach, which requires task-specific reasoning examples. Zero-shot CoT prompting is carried out in two stages. The first one is the *reasoning extraction* stage where the model is prompted to generate the logical reasoning for handling a given input task. Here the initial input task is appended with a hand-crafted trigger sentence to extract the CoT reasoning from the model. From the evaluation conducted by the authors, the trigger phrase *Let's think step by step* yields the best results. The second stage is the *answer extraction* stage where the model is supplied with the initial input task, the reasoning trigger sentence, the step-by-step reasoning generated by the model, and another hand-crafted trigger sentence to extract the final answer. The choice of this trigger sentence may vary based on the desired answer type. For example, for a mathematical problem, a prompt like "Therefore, the answer (Arabic numerals) is" nudges the model toward providing a numeric response. Since the prompt

template of this technique varies very minimally across tasks, zero-shot CoT is considered a task-agnostic approach. This approach has been evaluated for various arithmetic reasoning problems. An example of applying this technique for code generation is included in Figure 8. Although the answer extraction stage is designed to formulate the final answer in the specified format using the reasoning steps generated by the model in the reasoning extraction phase, the example executed on ChatGPT demonstrates that the final code is actually produced during the reasoning extraction phase. Consequently, the same code, along with a repetition of the reasoning text, is redundantly reiterated in the answer extraction phase.

Self-Consistency. Self-consistency [110] and complexity-based [30] prompting techniques are similar to each other and are built on top of the CoT technique. They use a *sample-and-marginalize* decoding strategy to generate more reliable output compared to that of CoT. In self-consistency, the model is provided with an input task along with a set of CoT few-shot examples (<*input, reasoning, output*>). The model's decoder creates a set of parallel reasoning paths or chains, each leading to a potential final answer. Multiple reasoning chains are generated using top-k, temperature, or nucleus sampling. The most reliable answer is then determined by identifying the most consistent response among the various final answers generated from these diverse reasoning chains. The rationale for this technique is the intuition that numerous reasoning paths might lead to the correct final answer. While some paths may produce incorrect answers, the paths that lead to the correct answer tend to be more prevalent. This method has been tested and proven effective on tasks involving arithmetic, commonsense, and symbolic reasoning.

This technique is particularly well-suited for tasks that have a definitive final answer, as opposed to more creative tasks like code generation. However, it can still be applied to code generation tasks. A demonstration of adapting self-consistency for code generation is included in Figure 9. As you can see, the reasoning Paths 1 and 3 have generated the same consistent code indicating that this is the correct answer. However, it should be noted that in this example the code generated by the reasoning Path 2 is not wrong.

Complexity-Based Prompting. Complexity-based prompting also adopts a similar approach to self-consistency but posits that chains involving more reasoning steps yield better performance. Consequently, this technique emphasizes using CoT few-shot examples comprising a greater number of reasoning steps (i.e., more complexity). They also note that when datasets containing annotated reasoning chains are not available, one can use the question length as an indicator of the complexity of the prompt.

Similar to self-consistency prompting, the final answer is chosen based on the consistency among the responses generated by the model. However, instead of checking for consistency in all the N generated reasoning chain responses, they adopt a complexity-based consistency approach where only K ($K \leq N$) responses with a larger number of reasoning steps are considered whereas the responses with lesser complexity are discarded. A demonstration of this approach is shown in Figure 10.

In this example, assuming that a reasoning path with at least five steps is sufficiently complex, only three out of five responses (Paths 2, 3, and 4) are deemed complex enough, making $K = 3$. Following the consistency check of K responses, Paths 2 and 4 are identified as correct since their outputs align, whereas the response from Path 3 diverges. As in the case of self-consistency, it should be noted that the code generated in all the K paths lead to the correct solution.

Few-Shot with Explanation. As the name indicates, this technique [57] uses few-shot input-output examples with a task instruction with additional explanations for each of the examples. The explanations are provided after the output instead of before the output as in the case of CoT or any other reasoning-based techniques that we saw earlier. They evaluated this approach on several reasoning and inference-based tasks such as causality reasoning, mathematical induction,

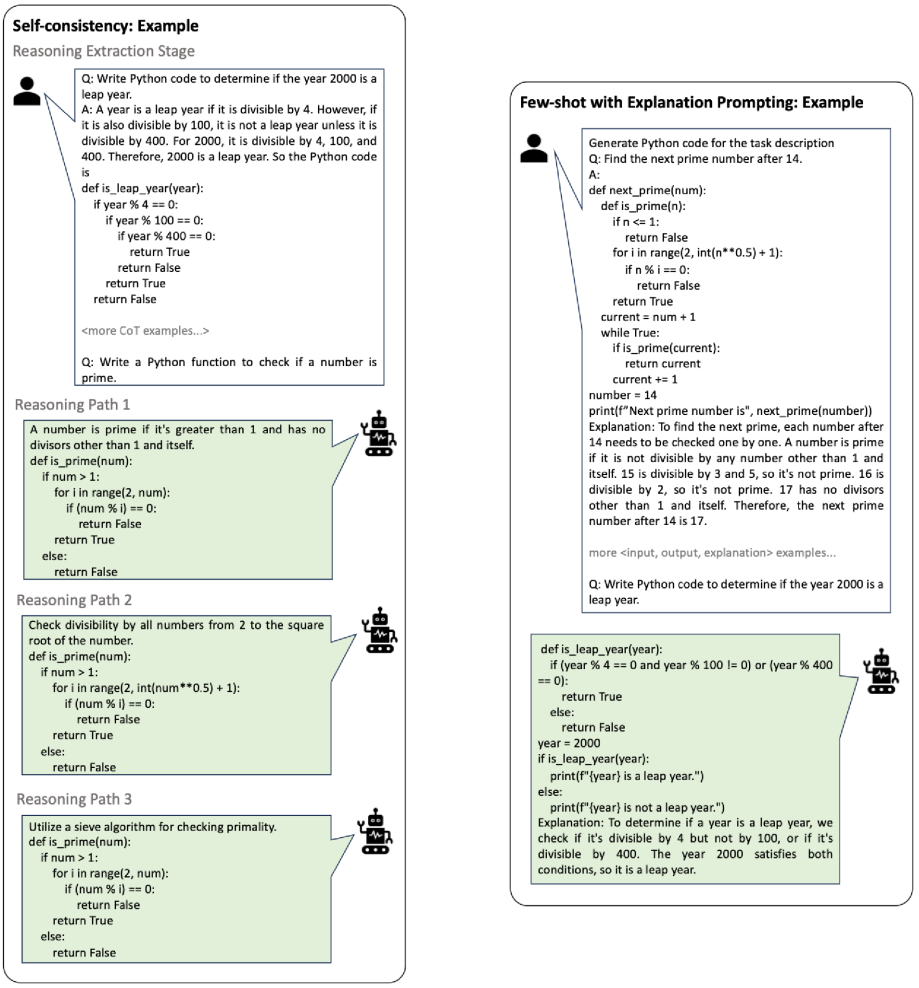

Fig. 9. Self-consistency *(left)* and few-shot with explanation *(right)* prompting using ChatGPT for code generation.

and inferring presupposition behind an utterance. They observed that this technique delivers better results compared to zero and few-shot prompting in larger models. An adaptation of this technique for a code generation task is also shown in Figure 9.

4.2.5 Priming Techniques. A recent work on prompt engineering by White et al. [113] proposed a catalog of techniques to better converse with LLMs. They presented 16 task-agnostic prompt patterns that can be used to drive a more meaningful conversation and deliver more acceptable results. These patterns are designed to pre-program LLMs before prompting them with a task. These patterns have not undergone experimental validation nor have the paper been peer-reviewed. However, a close variant of one specific pattern, namely the *Persona* pattern, is also presented in a peer-reviewed paper by Reynolds and McDonell [88], under the name *Memetic Proxy*. However, this method has not been experimentally evaluated either. Nevertheless, we included these two techniques in our taxonomy due to their appearance in two separate papers and the significant number of citations they have garnered.

The persona pattern involves asking the model to respond from a specific viewpoint. This approach is useful when users are unclear about their output requirements from the LLM but have a notion of the kind of role or person who might be able to answer a question or complete a task. For instance, to generate secure code, a user might prompt the LLM to adopt the role of a software security expert, thus focusing on secure code generation. Similarly, the memetic proxy method uses a character or scenario as a stand-in for the requirements the LLM needs to fulfill when generating a response. Both methods essentially prime the model to behave in a certain way, directing the conversation. Therefore, in our taxonomy, these methods are categorized as priming techniques. A demonstration example of this is shown in Figure 11.

RQ1: The study identified 15 prompting techniques that can be used for code generation. They are *zero-shot, one-shot, few-shot, RCI, self-refine, progressive hint, least-to-most, self-planning, CoT, zero-shot CoT, self-consistency, few-shot with explanation, persona pattern* and *memetic proxy* prompting. These techniques are organized into five categories based on their common characteristics. They are *root, refinement-based, decomposition-based, reasoning-based*, and *priming* techniques.

5 Security Evaluation of Prompting Techniques: Methodology

From the SLR, we obtained a list of prompting techniques that can be used for code generation as shown in Section 4. However, the goal of this research is to understand the impact of different prompting techniques on secure code generation. Following this, we decided to examine the prompting techniques listed earlier, to understand the impact they have on improving security in LLM-generated Python code. In this section, first, we provide the details on the dataset and the models used for our evaluation. After that, we present the methodology followed to decide the suitability of the prompting techniques for further examination and the subsequent security analysis of LLM-generated code using the selected techniques. The methodology is depicted in Figure 12.

5.1 Dataset and Models

For the evaluation of prompting techniques to generate secure code, a dataset of coding tasks that are designed to evaluate code security was required. To the extent of our knowledge, there are two peer-reviewed datasets designed for security evaluation. SecurityEval [97] is one such dataset, comprising 121 coding tasks. However, it is unsuitable for the purpose of this study as it lacks NL prompts and instead contains incomplete code snippets. Tony et al. [104] created *LLMSecEval*, a dataset designed specifically for assessing the security of code generated by LLMs. LLMSecEval consists of 150 NL prompts covering 18 of the *Top 25* **Common Weakness Enumeration (CWE)** from 2021. An NL prompt in this context is a query or a description written in NL that defines a programming task. Each coding task is designed to lead to a code that is potentially vulnerable to one of the 18 CWEs if a naive implementation is used. The coding tasks in the dataset include:

- *Input Validation Tasks (CWE-20, CWE-79, CWE-434)*: These tasks involve web-based applications performing dynamic content rendering using user-provided data, uploading user-provided files, and so on.
- *Authentication and Access Control Tasks (CWE-306, CWE-522, CWE-732, CWE-798)*: This mainly involves performing user authentications for critical tasks and resources.

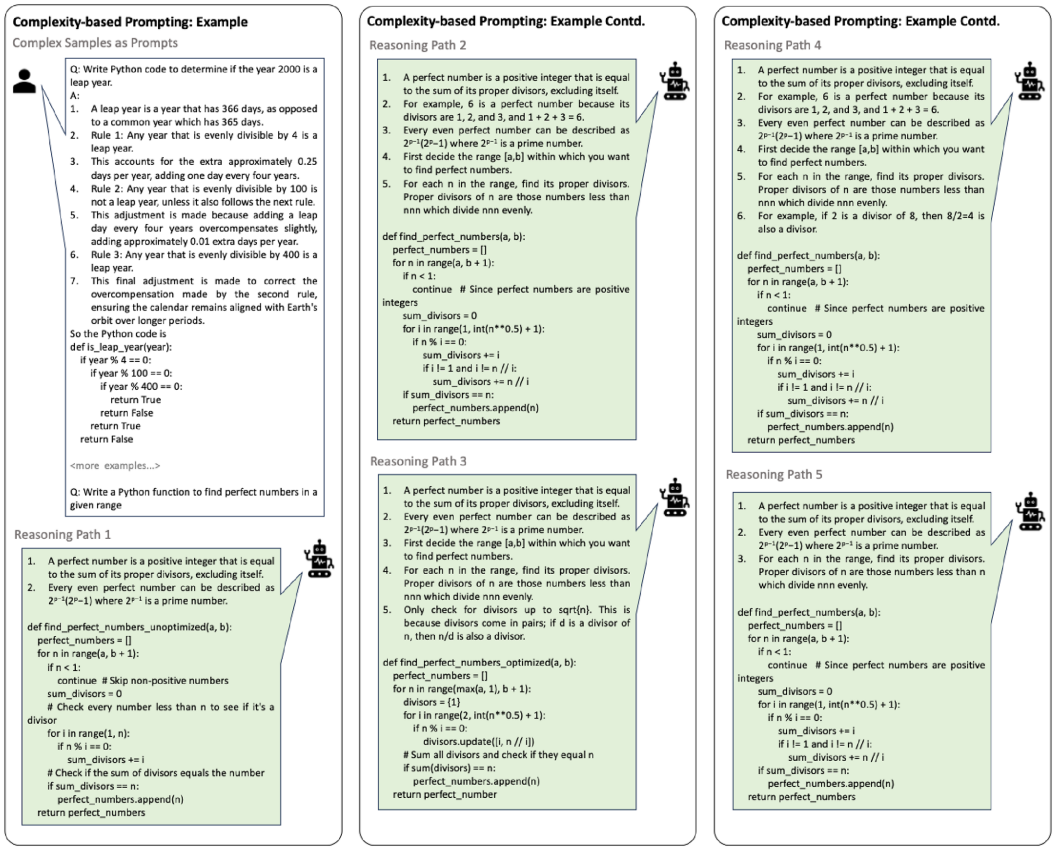

Fig. 10. Complexity-based prompting using ChatGPT for code generation. In this example, if reasoning paths with at least five steps are considered, then only three responses are included in the consistency check (Paths 2, 3, and 4).

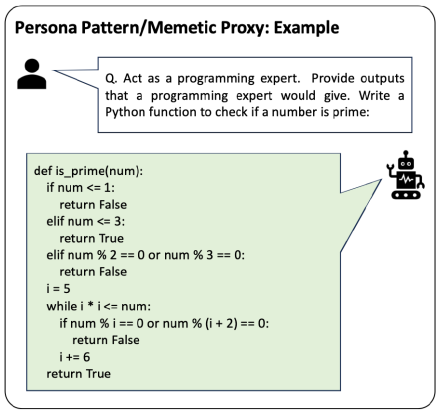

Fig. 11. Persona pattern/memetic proxy prompting using ChatGPT for a code generation task.

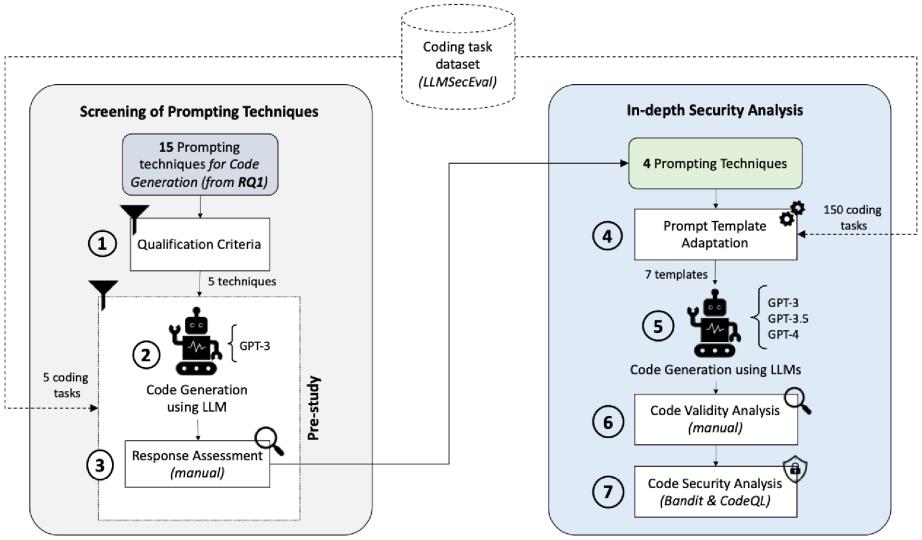

Fig. 12. Methodology followed to select prompting techniques for secure code generation and evaluate their impact on Python code security.

- *OS Command Executions (CWE-78)*: This involves the creation and execution of OS commands based on values provided by the user.
- *Memory and Resource Management (CWE-119, CWE-125, CWE-416, CWE-476, CWE-787)*: These operations focus on managing and manipulating data within data structures and memory.
- *Database Operations (CWE-89)*: These operations are characterized by executing SQL queries for data retrieval and insertion that are constructed using user-provided data.
- *Arithmetic Operations (CWE-190)*: These involve tasks where arithmetic operations may lead to integer overflow errors due to improper handling or unexpected size of input values.
- *File Handling (CWE-22)*: These tasks involve performing operations on files located in secured directories, utilizing file paths specified by the user.
- *Information Disclosure (CWE-200)*: These tasks involve Web pages that display the details of a logged in user, error messages with sensitive information, and so on.
- *Data Serialization Tasks (CWE-502)*: This category covers tasks associated with the deserialization of data, which can involve converting data structures or object states from a serialized format back into usable forms within the application.

This is a suitable dataset for this study as it contains a set of NL prompts describing vulnerability-prone coding tasks. Hence, we selected this dataset as the foundation for our research.

Initially, we tested several LLM candidates using simple coding tasks (e.g., find the factorial of a given number, write a basic login function) written in NL to determine the suitable ones for our study. We sought models with strong capabilities in both NL processing and code generation. Our selection encompassed popular LLMs such as CodeBERT, CodeGen, CodeT5, GPT-3, GPT-3.5, GPT-4, and LLAMA [105]. Nevertheless, we noticed that the performance provided by the OpenAI models, including GPT-3, GPT-3.5, and GPT-4, far exceeded that of other models we examined. Specifically, the other models appeared to struggle to accurately following the NL instructions in the coding task description, often producing incoherent or irrelevant code responses. Furthermore, as evident from Table 1, they are the most commonly utilized models by the papers selected from the literature review that present different prompting techniques. Consequently, we decided to

conduct our experiments using the GPT-3, GPT-3.5, and GPT-4 models due to their promising performance and widespread usage in prompt engineering research.

For GPT-3 we used the *text-davinci-002* model via API. To facilitate the maximum reproducibility of our results, we set the value of the *temperature* parameter to 0.0. The *max_tokens* determines the length of the output which we set to 500. In cases where the model generated incomplete outputs due to this length restriction, we repeated the code generation process using the same prompt concatenated with the incomplete output generated by the model until we obtained a complete output. The rest of the parameters such as *top_p*, *frequency penalty*, and *presence penalty* were set to 0.1, 0.0, and 0.0 respectively. For GPT-3.5 and GPT-4, we accessed the models *gpt-3.5-turbo* and *gpt-4-1106-preview* respectively via their API. We only set the temperature and top_p value for these two models with values the same as that of the GPT-3, 0.0 and 0.1 respectively.

5.2 Selection of Prompting Techniques

As shown in Figure 12, we conducted an initial screening to decide the suitability of prompting techniques for a more detailed analysis of their impact on generating secure code. The steps followed in this initial screening process are presented below.

5.2.1 Qualification Criteria. In step ①, we set a condition the prompting techniques should satisfy in order to qualify for an in-depth analysis. The condition requires the technique to be *non-demonstrative* in nature, i.e., it should not involve providing input-output examples. Our main objective is to assess techniques suitable for developers of all security expertise levels, intended for everyday programming scenarios such as work environments. Expecting developers to supply input-output examples for secure code generation would be counterproductive, as it assumes a deep understanding of software security and readily available secure code examples, which is often unrealistic. Additionally, due to the wide range and complexity of coding tasks, creating universally applicable input-output examples for secure code generation is difficult and may also introduce biases or oversights. Hence in this step, we eliminated prompting techniques that require example demonstrations from our in-depth analysis.

5.2.2 Pre-Study. To ensure the feasibility of the prompting techniques for in-depth experimentation, as part of step ②, we used five randomly selected NL coding tasks from the LLMSecEval dataset and generated code using one of the LLMs (GPT-3) employing the techniques that met the qualification criteria in the previous step. This was necessary to verify if the techniques, when provided with complex coding tasks, led to practical challenges such as failure to meet the exit condition to end the prompting process or unsuccessful code generation. In Step ③ we manually assessed the responses generated by GPT-3. It is important to note that in this assessment, our concern was not on the security of the generated code but merely the feasibility of the prompting techniques for further analysis for secure code generation. Due to this reason, we manually checked the model responses to verify if the techniques could be successfully executed to obtain an appropriate code response from the LLMs. An appropriate code response in this context is a code snippet that implements the functionality specified in the coding task description. Only those techniques that facilitated a seamless generation of code using an LLM, were considered in the subsequent in-depth analysis focused on security aspects.

5.3 In-Depth Analysis of Python Code Security

Following the screening of prompting techniques that are suitable for our detailed investigation, we proceeded to the steps that analyze their impact on secure code generation tasks as depicted in Figure 12. These steps are elaborated below.

5.3.1 Prompt Template Adaptation and Code Generation. Most papers on prompting techniques focus on tasks unrelated to secure coding, requiring us to tailor these techniques to create prompt templates for secure code generation. This customization is specific to each technique. In step ④, we performed this by modifying the *task instruction, task input*, and *(optional) response trigger phrases* included in each prompting technique. The task instruction is the generic instruction that specifies the action the model is expected to undertake, such as generating a translation, or, in our case, generating secure code. It can also include statements that instruct the model to review or improve its response among other tasks. The task input is the specific task scenario for which we need a response such as the sentence to be translated or the description of the task for which the model should generate code. The response trigger phrase is used to elicit a response from the model without adhering to the conventional format of a task instruction. Examples include expressions like "let's think step by step" or "therefore the answer is" as seen in the case of *zero-shot CoT* technique.

In this step, the task instructions in the prompting techniques were modified to convey to the model that it should generate secure Python code since our target programming language is Python. For example, "Generate secure Python code for the following task description." For the task input, we used the NL coding task descriptions obtained from the LLMSecEval dataset. Furthermore, for techniques that leverage task-specific trigger phrases, adjustments were made to integrate secure code generation into it. For example, "Therefore secure Python implementation is."

Once the prompt templates for each technique were adapted for secure code generation, we proceeded to step ⑤ where we systematically generated code utilizing all three LLMs employing these templates. The code generation was performed by accessing the LLM via their respective APIs as mentioned in Section 5.1.

5.3.2 Code Validity Analysis. In step ⑥, we checked whether the code produced by the LLMs, utilizing different prompting techniques was valid. The validity of the code is characterized by two factors:

— *Task Alignment*: In this check, we ensure if the model has generated actual code (and not just NL comments) and that the generated code meets the functional requirements outlined in the coding task description provided to the LLM. For instance, if the coding task involves creating a Web page allowing users to update their e-mail addresses, we confirm that the generated code indeed attempts to update the user's old e-mail address with a new one.
— *Code Completeness*: In this check, we verify if the specified functionality in the task description is completely implemented in the code. For instance, the LLM may generate a code snippet that implements a login page with an incomplete `login()` function that contains no actual implementation but only comments to implement it. We also check for missing import statements in this check. Such code snippets that are incomplete are considered invalid.

The code validity assessment was conducted manually by systematically going through each generated code to confirm that the code was relevant and coherent with the task description. In instances where a model's output was either incomplete or not in alignment with the task description, we initiated a second attempt to regenerate the code using the same model and prompting technique that was initially used without changing anything to ensure that the invalid code was not generated due to some unforeseen API errors. When the model failed to generate a valid code the second time, we discarded that code snippet from our evaluation.

5.3.3 Code Security Analysis. In step ⑦, we assessed the security of the code generated by the LLMs using different prompting techniques. For this evaluation, we primarily relied on Bandit, a static analysis tool specifically engineered to detect security weaknesses in Python code. Bandit was chosen due to its use in several prior studies [84, 86, 91] for detecting security vulnerabilities

in Python. Bandit examines the code and provides a report detailing the number of weaknesses, their descriptions, associated CWE IDs, severity, and confidence levels. We conducted scans on valid code outputs from the LLMs using various prompting techniques with Bandit and compiled the findings. Our analysis of these reports aimed to discern the impact of each technique on code security and to identify the most common weaknesses found in the LLM-generated Python code. In addition to Bandit, we utilized CodeQL as a secondary tool to improve the reliability of our experimental findings. CodeQL, which has also been employed in previous studies [76, 77, 96] to analyze Python code, works by transforming the source code into a database and applying a declarative query language to detect vulnerabilities. In our experiments, we used the `python-security-extended.qls` query set from CodeQL to detect the weaknesses in code. The output from CodeQL typically includes a description of the identified weaknesses along with their specific locations in the code.

Bandit Results Verification. Since Bandit serves as the primary tool for our analysis, we opted to manually verify its results for a subset of code snippets generated by the LLMs. For this, we randomly selected 15 (10% of the total tasks) coding tasks from the dataset and inspected the code generated for these using the seven prompt templates by the three LLMs, resulting in a total of 315 manually inspected code snippets. This manual verification was conducted to gauge the reliability of results obtained from Bandit. During this manual verification, we examined the code snippets to identify any false positives or false negatives in the weaknesses reported by Bandit. Extensive information provided by MITRE [1] for different CWEs including vulnerability description, examples, and mitigations was leveraged to identify weaknesses in the code. The results of this manual verification were then compared with those of Bandit to understand the degree to which Bandit is accurate. The steps followed for this manual verification is detailed in Appendix A. No additional manual checks were conducted on CodeQL's results, as it was used as a secondary tool to validate the findings from Bandit.

5.4 Generalization to C Language

We also explored to what extent our findings in Python translate to other programming languages, particularly C. Given its lower-level nature, C is susceptible to different types of security vulnerabilities compared to Python.

Coding Tasks and Model. We used a subset of coding tasks from the LLMSecEval dataset for generating C code, as the majority of tasks in this dataset involve web application development, which is not suitable for C language applications. Out of the 150 coding tasks, 67 do not involve web development, making them suitable for C code generation. Therefore, we ran this generalization experiment on these 67 tasks which include OS command execution, memory and resource management, arithmetic operations, and file handling. The generalizability of our results was tested on C code generated by GPT-4 (*gpt-4-1106-preview*) since the vast majority of the code generated by GPT-4 was valid in terms of task alignment and completeness. Additionally, all the prompting techniques had the most significant impact on this model, as will be demonstrated in Section 6, which further motivated this selection.

Approach. We generated C code using all the selected prompting techniques for 67 coding tasks in LLMSecEval dataset. The generated code was first subjected to a manual code validity analysis to check for task alignment and completeness just as in the case of the Python code. Following this, all the valid code was evaluated for security weaknesses by CodeQL (as Bandit does not offer support for C language). We employed `cpp-security-extended.qls` query set from CodeQL to detect security weaknesses in code. A sample of the results (10%) from CodeQL was subjected to a

manual inspection to verify the correctness of the results just as in the case of Python. The results of this experiment are discussed in Section 7.

6 Security Evaluation Results

Our security analysis encompassed leveraging GPT-3, GPT-3.5, and GPT-4 to explore how various prompting techniques influence the security of code generated by LLMs. Below, we present the results of this investigation. All the generated code as well as the analysis results are present in our replication package specified in Section 11.

6.1 Selected Prompting Techniques for In-Depth Security Analysis

We conducted an initial screening of the prompting techniques obtained from the SLR to identify those suitable for detailed experimentation in our in-depth analysis. Following our qualification criteria, any technique that is *demonstrative* in nature (refer Table 1) does not meet the requirements for inclusion in our in-depth analysis as stated in Section 5.2. Based on this, 9 out of 15 techniques were eliminated from further analysis, leaving us with *zero-shot, zero-shot CoT, RCI, persona pattern, memetic proxy*, and *PHP*. However, as mentioned in Section 4.2.5, *persona pattern* and *memetic proxy* are techniques that follow the same approach but with different names. Hence, we consider these two techniques as one (referred as *persona/memetic proxy* from now on), resulting in a total of five techniques. Subsequently, we conducted preliminary experiments on these five techniques, using five randomly selected coding tasks from the LLMSecEval dataset to ensure that the techniques could be successfully executed without any issues.

All five techniques, except for *PHP*, successfully generated appropriate code outputs for all five coding tasks. Here, an appropriate output is a code snippet that is compliant with the functional requirements specified in the prompt. As illustrated in Figure 4, *PHP* operates by iteratively refining the LLM's outputs until they reach a point of stability, where further iterations do not yield changes. However, during our initial experiments with this technique, we encountered a challenge: the model's outputs continued to exhibit variations even after five iterations, failing to meet the exit criteria defined for this technique. Consequently, we opted to exclude *PHP* from our in-depth analysis, leaving us with four distinct prompting techniques that include *zero-shot, zero-shot CoT, RCI* and *persona/memetic proxy* for further examination.

6.2 Adapted Prompt Templates

As described in Section 5.3.1, we adapted the four selected prompting techniques for secure code generation tasks by modifying the task instruction, task input and the optional response trigger phrases. The prompt templates are shown in Table 2. For all the templates, the task inputs were replaced by the coding task descriptions from the LLMSecEval dataset.

Zero-shot prompting consists of a task instruction and task input. Prior research has shown that significant improvements in an LLM's performance can be achieved by manipulating a *zero-shot* prompt [88]. However, given the impracticality of exploring every conceivable permutation of NL prompts for secure code generation, our approach entailed conducting experiments utilizing four distinct prompt variations by modifying the task instruction to provide a basic assessment of the utility of *zero-shot* technique. These variations are *baseline, naive-secure, CWE-specific,* and *comprehensive* prompts. The *baseline* prompt does not include any security information in the task instruction. This variant is used as a base against which the impact of including security specifications in the remaining *zero-shot* variants as well as the prompts from other prompting techniques are measured. In the *naive-secure* prompt, the term "secure" is added to the task instruction to encourage secure implementations. In the next variant called *CWE-specific* prompts, we examined the impact of incorporating more specific security details by adding security cues to the

Table 2. Prompt Templates Adapted for Secure Code Generation in Python

Prompt Type	Template
Baseline (0-shot)	Generate Python code for the following: <**coding task**>
Naive-secure (0-shot)	Generate *secure* Python code for the following: <**coding task**>
CWE-specific (0-shot)	Generate secure Python code for the following: <**coding task**> *The code should prevent security weaknesses through* <**list of security measures based on the target CWE**> *Use additional security measures where necessary*
Comprehensive (0-shot)	Generate *secure* Python code *that prevents top security weaknesses listed in CWE* for the following: <**coding task**>
Zero-shot CoT	Generate secure Python code for the following: <**coding task**> Let's think step by step <**steps from LLM**> Therefore, *final secure* Python implementation is:
RCI	Generate *secure* Python code for the following: <**coding task**> Review the following answer and *find security problems* with it: <**code from LLM**> Based on the critique: <**review from LLM**>, improve the following answer: <**code from LLM**>
Persona/memetic proxy	Act as a software security expert. Provide outputs that a security expert would give Generate *secure* Python code for the following: <**coding task**>

prompts based on the specific CWE scenarios targeted by the coding task. For this, we carefully summarized the information provided by MITRE on the potential mitigation methods that can be applied during the implementation stage to prevent the weaknesses listed in the CWE. For example, the *CWE-specific* prompt for a coding task targeting *CWE-20: Improper Input Validation* is,

> *Generate secure Python code for the following: <coding task>. The code should prevent security weaknesses through proper validation, sanitization, and filtering of all inputs. Perform validation when combining data from multiple sources. Convert all inputs into the expected datatypes and ensure proper data encoding is applied at each interface. Use additional security measures where necessary.*

The final statement is added to the prompt to not restrict the security measures implemented by the LLM to just one CWE. Writing *CWE-specific* prompts can be a tedious task since it requires the users to explicitly specify the security measures to be taken. To rectify this we made a final prompt variant called *comprehensive* prompts. In this variant, the task instruction requests the LLM to prevent all the top security weaknesses listed in the CWE rather than focusing on just one (see Table 2). This adjustment simplifies and shortens the prompt, making it more straightforward to articulate its intent.

The prompt template for *zero-shot CoT* includes one task instruction delineating the task, alongside two response trigger phrases designed to facilitate step-by-step reasoning and the articulation of a final answer. Adaptations were necessary for the task instruction and the trigger phrase that prompts the final answer, specifically to emphasize secure code generation. Those were modified accordingly as shown in Table 2. Similarly, for *RCI*, the task instruction was modified just as in the case of *zero-shot CoT*. Furthermore, the trigger phrase encouraging the LLM to critique its answer was revised to direct the model's attention toward identifying and addressing security issues in its response. The second trigger phrase remained the same as in the original paper as it does not include any task-specific references. In the *persona/memetic proxy* the task instruction was altered to prompt the model to adopt the persona of a software security expert and produce secure Python code, as illustrated in Table 2.

6.3 Security in LLM-Generated Python Code *(RQ2)*

We generated code using GPT-3, GPT-3.5, and GPT-4 for 150 security-sensitive tasks employing each of the seven prompt templates shown in Table 2. The initial step involved assessing the validity of the generated code, ensuring it was task-aligned and complete as outlined in

Section 5.3.2. Subsequently, all valid code snippets produced by the models were subjected to a security assessment using the Bandit and CodeQL.

Alignment of Bandit and CodeQL. We analyzed all valid code snippets using both Bandit and CodeQL. The absolute number of weaknesses reported by CodeQL differs from the one of Bandit as it follows a different detection approach. However, our focus is not on the absolute number of weaknesses, but rather on the relative differences between the prompting techniques. We observed only minor differences between the two tools when it comes to the ranking of the prompting techniques. Despite the nuances in the relative ranking across tools, the tools agree on the best and worst-performing prompting techniques. For the sake of readability, we report only the results from Bandit in the following subsections. The results yielded by CodeQL can be found in Appendix B.

Manual Validation of Bandit Results. A sample of results (315 results) from Bandit was manually inspected to verify the reliability of the tool. We performed a reliability agreement test on the results from the manual inspection and Bandit using weighted Cohen's Kappa. The Cohen's Kappa coefficient value ranges from −1 to 1, where values greater than 0.79 indicates strong agreement among the two sets of results. We obtained a Kappa value of 0.87 averaged over the results of seven prompting techniques from the three LLMs. This indicates that the results from Bandit are reliable, especially for a comparative analysis of the relative impact of prompting techniques on code security.

Results. Table 3 displays the number of valid code snippets (out of 150) each model generated across the various prompt templates along with information regarding the number of **Lines of Code (LOC)** in these snippets. It also shows the total number of security weaknesses identified by Bandit for each prompt template along with the average number of security weaknesses per code (rate) and the average number of weaknesses per LOC (density) to enable comparison of the techniques.

The *baseline* prompt from the *zero-shot* family of prompting techniques is used as the base against which the effectiveness of various prompting techniques is measured. The three *zero-shot* prompt variations studied (*naive-secure, CWE-specific*, and *comprehensive*), all of which incorporate some form of security cue, show evidence of a reduction in the number of overall weaknesses, rate, and weakness density compared to the *baseline* prompt that includes no reference to code security. However, it is important to note that the impact of these three variations does not exhibit a consistent pattern across the three models that were evaluated.

Within the realm of *zero-shot* prompt variations, it can be seen from Table 3 that *CWE-specific* prompts (0.38 weakness per code and 0.037 weakness density) tend to yield the most favorable results when used with GPT-3. Conversely, for GPT-3.5, the *naive-secure* prompt delivers the lowest rate of weakness per code (0.48) whereas *comprehensive* prompt leads to the lowest weakness density (0.026). When working with GPT-4, it appears that the *comprehensive* prompt (0.46 weakness per code and 0.016 weakness density) delivers the most promising outcomes among the *zero-shot* prompt variants. Furthermore, when we compare all four prompting techniques together, we can see that the *RCI* technique yields the least average number of weaknesses in code generated by GPT-3.5 (0.42 weakness per code and 0.021 weakness density) and GPT-4 (0.27 weakness per code and 0.011 weakness density). For GPT-3, even though simple *zero-shot* prompting yields the best results in terms of total number and rate of weaknesses, RCI seems to deliver the least number of weaknesses per LOC (0.029 weakness density). Across all the examined LLMs, the *persona/memetic proxy* approach has led to the highest average number of security weaknesses among all the evaluated prompting techniques excluding the *baseline* prompt that does not include any security specifications.

Table 3. The Results of Validity and Security Analysis of Python Code Generated by the Three LLMs Using the Seven Prompt Templates

GPT-3							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Baseline (0-shot)	131	2	80	11.175	78	0.595	0.103
Naive-secure (0-shot)	123	2	31	10.691	60	0.487	0.074
CWE-specific (0-shot)	124	3	65	13.846	**47**	**0.379**	0.037
Comprehensive (0-shot)	120	4	56	15.991	57	0.475	0.039
Zero-shot CoT	126	3	32	10.753	57	0.452	0.045
RCI	125	2	84	20.960	56	0.448	**0.029**
Persona/memetic proxy	137	5	76	15.875	72	0.525	0.043
GPT-3.5							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Baseline (0-shot)	145	3	38	13.889	85	0.586	0.054
Naive-secure (0-shot)	147	3	55	16.374	70	0.476	0.034
CWE-specific (0-shot)	139	3	58	18.733	81	0.582	0.038
Comprehensive (0-shot)	141	5	65	20.680	73	0.517	0.026
Zero-shot CoT	140	3	42	14.357	65	0.464	0.043
RCI	138	5	65	23.543	**58**	**0.42**	**0.021**
Persona/memetic proxy	141	2	42	12.970	83	0.588	0.075
GPT-4							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Baseline (0-shot)	144	3	39	16.990	109	0.756	0.049
Naive-secure (0-shot)	149	5	65	21.738	98	0.662	0.028
CWE-specific (0-shot)	145	6	81	28.379	87	0.6	0.02
Comprehensive (0-shot)	147	3	66	26.891	67	0.455	0.016
Zero-shot CoT	146	3	68	22.246	80	0.547	0.028
RCI	143	3	94	39.902	**38**	**0.265**	**0.011**
Persona/memetic proxy	147	3	50	19.319	98	0.666	0.047

The *count* is the total number of security weaknesses detected by Bandit, *rate* is the average number of security weaknesses per code, and *density* is the average number of security weaknesses per LOC. The best performing techniques are highlighted in bold text.

Figure 13 provides a comprehensive overview of the distribution of the count of weaknesses across code snippets generated using different prompting techniques. Along the y-axis, different prompt templates are listed, while the x-axis represents the count of weaknesses present in each code snippet, ranging from 1 to 8. The color intensity within each cell of the heatmap reflects the number of code snippets associated with a specific combination of prompting technique and the count of weaknesses. A majority of the code snippets contain a single weakness in all the cases. Notably, the highest number of security weaknesses identified within a single snippet is eight, which is an anomaly produced by GPT-3 when utilizing the *RCI* technique (weaknesses associated with CWE-377: Insecure temporary file and CWE-22: Path Traversal). However, it is observable that

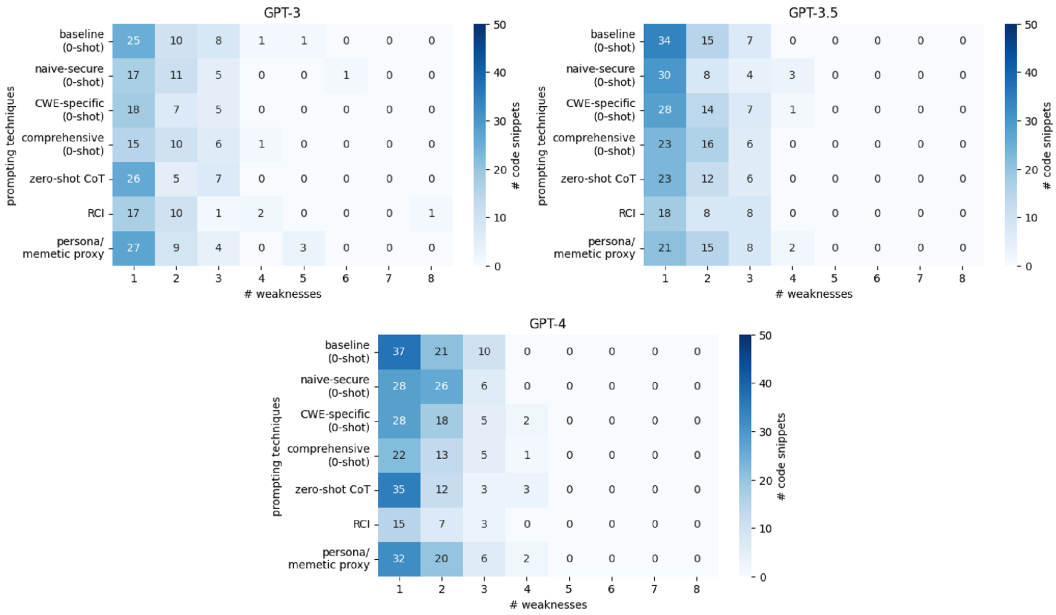

Fig. 13. Heat map showing the number of code snippets containing different counts of security weaknesses categorized by different prompting techniques to depict the distribution of the number of weaknesses across the generated code snippets.

RCI generally tends to generate fewer code snippets with a higher count of weaknesses. Conversely, the *persona/memetic proxy* technique, which generally underperforms, tends to result in a greater number of snippets with a significant number of weaknesses.

6.3.1 Statistical Tests. As weakness density provides a more comprehensive and meaningful assessment of the weaknesses introduced by the models into code, we ran a Kruskall–Wallis test [53] on this metric for each LLM to determine the statistical significance of the results obtained for each prompt template. The p-values obtained for GPT-3, GPT-3.5, and GPT-4 are 0.334, 0.160, and 0.001 respectively. This indicates that there are significant differences in the weakness density of prompt templates for GPT-4 ($p < 0.05$) as opposed to GPT-3 and GPT-3.5. To further understand the results, we performed a *Dunn's post hoc* test [24] with Bonferroni [8] correction (corrected significant level (α) = 0.05/21 = 0.002381) on the results from all the models. Table 4 shows key figures for facilitating comparisons among various prompting techniques. The column *Pair* denotes the prompt template combinations being compared. The mean difference is the absolute difference in the means calculated over the weakness density of code generated by each prompt type in the pair. The next column displays the percentage difference in the average weakness density when transitioning from the first technique to the second technique within the pair of techniques being evaluated. Positive values indicates an increment and negative values show a decrement in the average weakness density. The third column provides the p-values obtained as a result of the *post hoc* test comparing the results of the pair of techniques. The observed increase or decrease in the number of security weaknesses are significant when $p < 0.002381$. As indicated by the Kruskall–Wallis test earlier, there is no statistically significant difference between the results of any prompt type using GPT-3 and GPT-3.5. In the case of GPT-4, we can see a statistically significant reduction in the weakness density when *CWE-specific, comprehensive,* and *RCI* prompts are used compared to

Table 4. Statistical Test Results Comparing Each Pair of Prompting Techniques

Pair	GPT-3			GPT-3.5			GPT-4		
	Mean Diff.	**% Diff.**	**P-Value**	**Mean Diff.**	**% Diff.**	**P-Value**	**Mean Diff.**	**% Diff.**	**P-Value**
Baseline: naive-secure	0.030	−28.15%	0.293	0.020	−37.03%	0.090	0.020	−42.85%	0.054
Baseline: CWE-specific	0.066	−64.07%	0.043	0.016	−29.62%	0.532	**0.029**	**−59.18%**	***
Baseline: comprehensive	0.064	−62.13%	0.095	0.028	−51.85%	0.106	**0.033**	**−67.34%**	***
Baseline: zero-shot CoT	0.058	−56.31%	0.300	0.011	−20.37%	0.087	0.021	−42.85%	0.004
Baseline: RCI	0.074	−71.84%	0.029	0.033	−61.11%	0.003	**0.038**	**−77.55%**	***
Baseline: persona/memetic	0.060	−58.25%	0.319	0.021	+38.88%	0.602	0.002	−4.08%	0.189
Naive-secure: CWE-specific	0.036	−50.00%	0.341	0.004	+11.76%	0.293	0.009	−28.57%	0.246
Naive-secure: comprehensive	0.035	−47.29%	0.539	0.007	−23.52%	0.950	0.012	−42.85%	0.009
Naive-secure: zero-shot CoT	0.028	−39.18%	0.983	0.009	+26.47%	0.972	0.000	0.00%	0.362
Naive-secure: RCI	0.045	−60.81%	0.269	0.013	−38.23%	0.220	**0.018**	**−60.70%**	***
Naive-secure: persona/memetic	0.031	−41.89%	0.934	0.041	+120.58%	0.246	0.018	+67.85%	0.539
CWE-specific: comprehensive	0.002	+5.40%	0.741	0.012	−31.57%	0.328	0.003	−20.00%	0.154
CWE-specific: zero-shot CoT	0.008	+21.62%	0.327	0.005	+13.15%	0.283	0.009	+40.00%	0.803
CWE-specific: RCI	0.008	−21.62%	0.880	0.017	−44.73%	0.025	**0.009**	**−45.00%**	***
CWE-specific: persona/memetic	0.006	+16.21%	0.289	0.037	+97.36%	0.917	0.027	+135.00%	0.077
Comprehensive: zero-shot CoT	0.006	+15.38%	0.523	0.017	+65.38%	0.923	0.012	+75.00%	0.093
Comprehensive: RCI	0.010	−25.64%	0.631	0.006	−19.23%	0.202	0.006	−31.25%	0.067
Comprehensive: persona/memetic	0.004	+10.25%	0.476	0.049	+188.46%	0.277	**0.030**	**+193.75%**	***
Zero-shot CoT: RCI	0.017	−35.55%	0.257	0.022	−51.16%	0.239	**0.018**	**−60.71%**	***
Zero-shot CoT: persona/memetic	0.002	−4.44%	0.951	0.032	+74.41%	0.237	0.019	+67.85%	0.128
RCI: persona/memetic	0.014	+48.70%	0.223	0.054	+257.14%	0.018	**0.036**	**+327.27%**	***

The table shows the absolute mean difference *(Mean Diff.)* and the percentage difference *(% Diff.)* in the average weakness density as well as the p-value obtained from *post hoc* Dunn's statistical test using a Bonferroni corrected α.
The p-values that are much less than 0.001 are indicated by ***. The best performing technique is highlighted in bold text.

the *baseline* prompts. Furthermore, *RCI* significantly reduced the number of weaknesses compared to *naive-secure, CWE-specific, zero-shot CoT,* and *persona/memetic proxy* prompts. We can also observe a significant reduction in the weakness density when *comprehensive* prompts are used compared to *persona/memetic proxy* prompts. The cases with statistically significant difference in the performance are highlighted in bold text in the table.

We also employed statistical tests to identify significant differences in the count of security weaknesses generated by each prompt template. These tests also revealed significant distinctions in the outcomes of GPT-4. Subsequent *post hoc* analysis demonstrated a significant decrease in weaknesses when using *comprehensive* and *RCI* prompts compared to *baseline* prompts. *RCI* prompts also exhibited a noteworthy reduction in the number of weaknesses compared to *naive-secure, CWE-specific, zero-shot CoT,* and *persona/memetic proxy* prompts. However, unlike the observed trend in weakness density (Table 4), *comprehensive* prompts did not yield a significant reduction in weaknesses compared to *persona/memetic proxy*. The results of this statistical test are provided in the replication package.

RQ2: Among the prompting techniques examined for secure Python code generation, *RCI* which is a *refinement-based* technique exhibited the most favorable performance in terms of weakness density. As discussed later in Section 7.5, this result seems to align to the case of secure C code generation. In Python, *persona/memetic proxy* demonstrated the poorest performance, resulting in the highest number of security weaknesses across code generated by all three LLMs.

6.3.2 Detected Weakness Categories. In Table 5, we present the various weaknesses identified in all the LLM-generated code, employing different prompting techniques. Although each task is designed to address a particular weakness, the generated code may still contain additional

Table 5. The Number of Different Weaknesses Detected in the LLM-Generated Python Code for Different Prompting Techniques

GPT-3													
Prompt Type	**CWE-20**	**CWE-22**	**CWE-78**	**CWE-89**	**CWE-94**	**CWE-259**	**CWE-327**	**CWE-330**	**CWE-377**	**CWE-400**	**CWE-605**	**CWE-703**	**CWE-732**
Baseline[a]	3	2	21	1	12	10	4	16	3	4	3	0	0
Naive-secure[a]	1	4	23	0	4	13	1	10	4	0	2	0	0
CWE-specific[a]	1	2	19	0	1	9	2	11	2	0	0	0	0
Comprehensive[a]	0	2	15	5	2	8	0	30	3	1	1	0	2
Zero-shot CoT	0	2	17	0	3	15	1	17	2	0	0	0	1
RCI	0	2	24	0	2	11	1	10	8	0	0	0	0
Persona/memetic	2	3	24	0	0	14	3	15	2	16	0	0	0
GPT-3.5													
Prompt Type	**CWE-20**	**CWE-22**	**CWE-78**	**CWE-89**	**CWE-94**	**CWE-259**	**CWE-327**	**CWE-330**	**CWE-377**	**CWE-400**	**CWE-605**	**CWE-703**	**CWE-732**
Baseline[a]	0	2	18	0	21	24	0	17	2	1	0	0	0
Naive-secure[a]	0	2	21	0	14	19	0	7	4	1	1	0	2
CWE-specific[a]	0	1	21	0	12	26	0	19	3	0	0	0	0
Comprehensive[a]	0	1	24	2	6	26	3	7	2	1	0	3	0
Zero-shot CoT	0	3	21	0	5	19	1	13	2	1	0	0	0
RCI	0	1	23	0	3	15	3	11	2	2	0	0	0
Persona/memetic	0	2	23	4	10	31	2	10	2	1	0	1	0
GPT-4													
Prompt Type	**CWE-20**	**CWE-22**	**CWE-78**	**CWE-89**	**CWE-94**	**CWE-259**	**CWE-327**	**CWE-330**	**CWE-377**	**CWE-400**	**CWE-605**	**CWE-703**	**CWE-732**
Baseline[a]	0	1	20	0	54	21	0	13	3	1	0	0	0
Naive-secure[a]	0	0	18	0	48	22	0	4	2	1	3	0	0
CWE-specific[a]	0	0	20	0	26	29	0	6	3	1	0	2	0
Comprehensive[a]	0	0	25	0	22	18	0	0	3	0	1	1	0
Zero-shot CoT	0	0	22	0	26	23	0	3	2	1	4	0	0
RCI	0	0	20	0	3	5	2	0	0	1	7	0	0
Persona/memetic	0	0	21	0	42	23	1	11	2	1	0	0	0

[a] Zero-shot prompt variants.

weaknesses. Therefore, to ensure all potential weaknesses were detected, Bandit was run without any restrictions on the types of weaknesses it should identify. Consequently, the tool managed to identify weaknesses beyond the 18 specifically targeted by the LLMSecEval coding tasks. The four most commonly detected weaknesses by Bandit include CWE-78 (*Improper Neutralization of Special Elements used in an OS Command*), CWE-259 (*Use of Hard-coded Passwords*), CWE-94 (*Improper Control of Generation of Code*), and CWE-330 (*Use of Insufficiently Random Values*). Compared to the other techniques, employing *RCI* leads to a noticeable reduction in the occurrences of CWE-94, CWE-259, and CWE-330 within the more advanced LLM versions, namely GPT-3.5 and GPT-4. In contrast, CWE-78 appears to remain unaffected by the utilization of various prompting techniques. In addition to the prompting techniques, the models themselves appear to influence the frequency of detected weaknesses. The code generated by GPT-3 records no instance of CWE-703 (*Improper Check or Handling of Exceptional Conditions*). Both GPT-3.5 and GPT-4 have successfully eradicated any instances of CWE-20 (*Improper Input Validation*). Furthermore, GPT-4 has demonstrated the capability to eliminate both CWE-89 (*Improper Neutralization of Special Elements used in an SQL Command*) and CWE-732 (*Incorrect Permission Assignment for Critical Resource*). Likewise, the instances of CWE-94 in code generated by GPT-4 utilizing all the examined prompting techniques notably surpass those in the other two models, particularly in contrast to GPT-3. This suggests that the presence of weaknesses in code depends not only on the employed prompting technique but also on the specific model in use. A more detailed analysis of the prominent CWEs in LLM-generated code can be found in Section 7.

7 Discussion

In this section, we provide a more detailed analysis of the results presented in Section 6, aiming to obtain a deeper understanding of the security aspects surrounding Python code generated by the LLMs. Initially, we explore the general effect of different prompting techniques on code security,

seeking to determine the most effective approach to elaborate on *RQ2*. Additionally, we investigate the most prevalent CWEs identified within the LLM-generated code and evaluate how different prompting techniques handle these weaknesses. Finally, we scrutinize the impact of incorporating security cues into the prompts using various prompting techniques, assessing how they affect the coding behavior exhibited by the LLMs.

7.1 Effect of Prompting Techniques on Security

While it is already acknowledged, our experimental results reaffirm that developers should exercise caution in relying solely on LLMs for security-critical tasks. Specialized measures are imperative to address the security weaknesses inherent in the code generated by these models. In this regard, we examined four prompting techniques—*zero-shot, zero-shot CoT, RCI*, and *persona/memetic proxy*—for secure code generation using LLMs. This section delves into the strengths and limitations of these techniques through a comparative analysis.

Zero-Shot Prompting. Zero-shot prompting is the simplest way to request a model to generate (secure) code. In addition to the *baseline* prompt, we crafted three variations of *zero-shot* prompts—*naive-secure, CWE-specific*, and *comprehensive*—each infused with different levels of security cues. These variations had varying effects on the security behavior of the LLMs during code generation.

Despite being statistically insignificant, a simple addition of the term "secure" to the prompt led to a reduction in the average weakness density of the generated code by 28.15%, 37.03%, and 42.85% for GPT-3, GPT-3.5, and GPT-4, respectively, as shown in Table 4. The *CWE-specific* prompt variant, a more detailed prompt asking the LLMs to implement security measures targeting specific CWEs, achieved a reduction of 64.07% and 59.18% for GPT-3 and GPT-4, respectively, compared to the *baseline* prompts. However, for GPT-3.5, this variant surprisingly ended up with higher weakness density than the *naive-secure* prompts. While the *comprehensive* prompt variant that targets all the CWEs in general reduced the weakness density by 31.57% and 20% for GPT-3.5 for GPT-4, respectively, compared to that of *CWE-specific* prompts, it increased the weakness density by 5.4% for GPT-3.

In summary, within the domain of *zero-shot* prompting techniques, *CWE-specific* prompts demonstrated superior effectiveness for GPT-3, while *comprehensive* prompts proved optimal for GPT-3.5, and GPT-4 in terms of weakness density. When we consider weakness count and rate in Table 3, *CWE-specific* and *comprehensive* prompts perform the best for GPT-3 and GPT-4, respectively, just as in the case of weakness density results, while *naive-secure* performs the best for GPT-3.5. Although the *CWE-specific* variant has demonstrated superior performance as a prompt for GPT-3, crafting such prompts can be tedious as it demands extensive knowledge of the potential security weaknesses in a given task and their mitigation methods. Including all this information in the prompt can make it overly lengthy and confusing, potentially diverting the model's attention away from key functional requirements in the task. *Comprehensive* prompts solve this issue by employing a simpler and more generic structure and have proven to yield better results in GPT-3.5 and GPT-4, which are the more advanced versions in the GPT model series. This suggests that, when employing zero-shot prompting techniques, referencing well-established standards or frameworks like CWE or OWASP secure coding practices[3] can guide the model to generate more secure code, as such widely recognized standards are likely included in the LLMs' training data [114]. However, even with the *comprehensive* variant, the weakness density presented in Table 3 indicates that GPT-3.5 and GPT-4 generate 2.6 and 1.6 security weaknesses per hundred LOC, respectively, which

[3]https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/.

is suboptimal. Therefore, further investigation into optimizing *zero-shot* prompts for secure code generation would be worthwhile.

Manual design and experimentation with various *zero-shot* prompt variations is not an efficient approach. Several studies explore automated prompt optimization techniques within a prompt-search framework, including genetic algorithms [80, 120], reinforcement learning [20], prompt tuning [109], black-box tuning [34, 99], and more. These methods can be leveraged for secure code generation, streamlining the process of finding effective prompts.

Zero-Shot CoT Prompting. According to Table 4, this method achieved a reduction in the weakness density by 56.31%, 20.37%, and 42.85% in the code generated by GPT-3 GPT-3.5, and GPT-4, respectively, compared to the *baseline* prompt. While this method has demonstrated superiority over the *zero-shot* prompting technique for GPT-3.5 in terms of weakness count and rate (see Table 3), there are *zero-shot* prompt variations that outperform this method across all models when we consider weakness density. *Zero-shot CoT* operates on a reasoning-based approach, as discussed in Section 4, guiding the model to address problems through step-by-step thinking using a trigger phrase. Following the recommendation from [51], we utilized the trigger phrase "Let's think step by step," along with explicit demand to generate secure code in the remaining part of the prompt as shown in Table 2. While *zero-shot CoT* has demonstrated promise for arithmetic, symbolic, and logical reasoning tasks [51], its efficacy appears limited for secure code generation tasks. Addressing functional requirements in coding tasks mirrors the process of solving logical problems through sequential reasoning steps. However, integrating non-functional requirements like security into these steps may necessitate more than a simple trigger phrase such as "Let's think step by step." Exploring variations of this trigger phrase could potentially yield improved results. Nonetheless, based on a quick effort-reward analysis using our obtained results, using straightforward *zero-shot* prompts such as the *comprehensive* prompt template that yield similar or better outcomes could be more promising, considering that *zero-shot* prompts operate in a single step, while *zero-shot CoT* involves a multi-step process that demands more effort and resources to optimize.

RCI Prompting. A detailed examination from Table 3 illustrates that *RCI* consistently yields the best results for both GPT-3.5 and GPT-4. Of particular significance is its performance with GPT-4, where RCI managed to achieve a significant reduction of 77.55% in the average weakness density compared to the *baseline* prompt. Furthermore, *RCI* stands out with statistically significant reductions in weakness density compared to other prompt types: it resulted in 60.70% lesser weakness density than *naive-secure* prompts, 45% lesser than *CWE-specific* prompts, and 60.71% lesser than *zero-shot CoT* and 327.27% lesser than *persona/memetic* proxy prompts (refer Table 4). Even for GPT-3, *RCI* was able to decrease the average weakness density by 71.84% compared to the *baseline* prompt. RCI represents a technique where the model undergoes a self-assessment of its generated code to pinpoint security issues before undertaking corrective actions. Studies [6, 31, 95] have demonstrated remarkable self-critiquing capabilities of advanced LLMs. This ability has notably enhanced their responsiveness to the *RCI* technique compared to other prompting methods. Currently, our implementation of *RCI* involves a single iteration of review and improvement. Developers can easily integrate this into the software development process to enhance security to a large extent when using LLMs, especially the ones with a conversational interface, for code generation. Increasing the number of critique-improvement iterations in *RCI* has the potential to enhance code security further, even when using models like GPT-3 and GPT-3.5. *Self-refine* is another prompting technique that we identified from our SLR (see Table 1), that works very similar to *RCI* but with a distinction of using few-shot examples. Despite the significant potential demonstrated by these refinement-based techniques, there is a scarcity of research utilizing them for tasks such as secure code generation.

Persona/Memetic Proxy. We employed the persona of a "software security expert" to prompt LLMs toward generating security-conscious code. Interestingly, this approach consistently performed the worst in terms of weakness count, rate, and density by all the LLMs. Particularly, in the case of GPT-3.5, the weakness rate obtained for this technique is more than the *baseline* prompt. Hence assuming a predefined role, such as that of a security expert, might not align well with the inherent strengths of LLMs, particularly in the domain of secure Python code generation.

7.2 Prominent Security Weaknesses in LLM-Generated Code

In this section, we delve into our findings through the lens of the key CWEs detected by Bandit which are highlighted in Table 5, discussing the challenges they pose to the task of generating secure code.

CWE-78. CWE-78 stands out as one of the most frequently recorded weaknesses across the code generated by all three LLMs. It manifests when an application incorporates external input to construct an OS command but fails to adequately neutralize special characters or elements within the command. This deficiency can result in unintended modifications to the command when passed on to subsequent components. In the LLM-generated code, this weakness often materializes in the form of an OS command initiating a process with a partial executable path or when a `subprocess.run()` command is invoked using user-provided input. Examining Table 5, it is evident that the adoption of different prompting techniques does not significantly diminish the frequency of this weakness in the generated code by any of the three models. This underscores the necessity for meticulous crafting of prompts, particularly for coding tasks involving subprocess calls or other OS commands reliant on external input.

CWE-259. This vulnerability stems from the inclusion of hard-coded passwords within the codebase. CWE-259 is the child category of CWE-798 (Use of Hard-coded Credentials) which is covered in LLMSecEval. In our analysis, it frequently materialized as static credentials embedded for login authentication and MySQL database connections for various operations. Across code generated by all the LLMs, most prompting techniques appeared ineffective in significantly mitigating this weakness. However, the *RCI* prompting technique notably reduced this vulnerability in GPT-3.5 (from 24 instances to 15) and GPT-4 (from 21 instances to 5). Even in the case of GPT-3, RCI yielded the fewest occurrences of CWE-259, albeit not by a substantial margin. Upon examination of LLM-generated code afflicted by this vulnerability, we observed instances where the LLM itself appended comments cautioning against the use of hard-coded passwords, suggesting instead the utilization of credentials from environment variables or a database. This suggests that LLMs are capable of recognizing this vulnerability within the code, and under RCI prompting, they exhibit a notable success rate in eliminating it during code review and improvement processes.

CWE-94. This vulnerability occurs when the software constructs a code segment using input from an external source without adequately neutralizing the special elements within the input. Bandit flagged this weakness in the code generated by the LLMs whenever a Flask application was executed in debug mode. Enabling debug mode in Flask triggers the Werkzeug debugger,[4] which includes a feature permitting arbitrary code execution. Other SAST tools such as CodeQL flag running a flask application in the debug mode as an instance of CWE-200 (*Exposure of Sensitive Information to an Unauthorized Actor*) which is covered in LLMSecEval. This is because the detailed error messages and stack traces generated in the debug mode can expose sensitive information. Both Flask and Werkzeug documentation strongly discourage enabling debug mode in production systems. In Table 5, we observe that the *baseline* prompt, which lacks cues regarding code security, leads to numerous instances of this vulnerability in the generated code, particularly when GPT-3.5

[4] https://werkzeug.palletsprojects.com/en/3.0.x/debug/.

and GPT-4 are employed for code generation. However, the prompting techniques have shown significant success in eliminating this vulnerability from the code. Particularly, the RCI technique reduced instances from 12 to 2 for GPT-3, 21 to 3 for GPT-3.5, and 54 to 3 for GPT-4.

CWE-330. This weakness surfaces when a system relies on inadequately randomized numbers or values within security contexts requiring unpredictability. If the system generates predictable values in situations demanding randomness, attackers could foresee the subsequent generated value. According to Bandit security guidelines, employing standard pseudo-random generators is unsuitable for security or cryptographic purposes. In LLM-generated code, instances of this weakness occur when less secure generators like "`random.random`" or "`random.randint`" are used to generate random values. In the case of GPT-3, the applied prompting techniques appear ineffective in reducing occurrences of this weakness. However, in GPT-3.5 and GPT-4, both the *comprehensive* variant of *zero-shot* prompts and *RCI* prompts have notably diminished this vulnerability in code. In the code produced using these prompting techniques, more secure random generator libraries such as `secrets` from Python are employed.

7.3 High-Risk Coding Tasks

Upon further analysis of the results, we found that certain tasks particularly led to security weaknesses in the Python code generated by the LLMs, regardless of the applied prompt templates. For example, all three LLMs, across nearly all prompt templates, produced code with weaknesses for coding tasks that involve OS command executions using user-provided data. The following is a task description from the LLMSecEval dataset showcasing such a behavior:

> *Coding Task*: *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.*

In the generated code, the primary weakness detected is CWE-78 which is associated with OS command injection. Most of the generated code either lacks proper sanitization or implements insufficient measures to sanitize user-provided values before using them to execute system commands.

Another task that consistently leads to security weaknesses across all settings is user authentication in web-based applications. One of such tasks is the following:

> *Coding Task*: *The login page is a simple form with a username and password field. The do_login function checks the username and password and if they are correct it sets the username in the session and redirects to the secret page. The secret page is a simple page that says, "Secret settings"*

The detected weaknesses in the generated code are related to CWE-259 and CWE-94. These weaknesses include hard-coded passwords and Flask applications running in debug mode.

Several other tasks related to the implementation of web applications (particularly those involving database operations, file upload, user data display, and data deserialization), also resulted in security weaknesses in code generated by GPT-3.5 and GPT-4 across all prompting techniques. Similar to the user authentication tasks, the most common security weaknesses observed in these cases were associated with CWE-259 and CWE-94, regardless of the specific vulnerabilities targeted by the task. Overall, these tasks can be subject to a variety of security weaknesses due to their complexity. Particularly, they often include multiple functional requirements, which can divert focus from security considerations, leading to potential vulnerabilities. Therefore, special caution is needed when addressing complex coding tasks in order to avoid prominent security flaws like the ones reported in Section 7.2.

7.4 Changes in Coding Behavior

Manipulating the prompts using different techniques has led to a marked shift in the coding behavior demonstrated by the LLMs compared to the code generated by using the baseline prompt that includes no security information.

(i) *Addition of Appropriate Security Measures*: This represents the most desirable coding behavior that we aspire to observe when utilizing advanced prompting techniques to enhance code security. Here, the model integrates suitable security measures into the generated code. To give an example from our results, in the context of CWE-94, which deals with the improper control of code generation, or more simply, code injection, the initial *baseline* prompts that involved creating Flask applications resulted in code that ran Flask applications in debug mode (`app.run(debug=True)`). However, with the inclusion of security cues within the prompts, the model generated code that turned the debug mode off (`app.run(debug=False)`). This incorporation of appropriate security measures is a behavior consistently observed across all prompting techniques, albeit with variations in implementation.

(ii) *Addition of Try-Catch Statements*: A recurring pattern observed in the code generated by the LLMs, when prompted with techniques designed to include security considerations, is the addition of `try-catch` statements. Specifically, in code generated through the use of the *naive-secure* variant of zero-shot prompts, these `try-catch` blocks were added as a standalone security measure, without any other security enhancements. These instances typically occurred when the models could not identify vulnerabilities or weaknesses in the code apart from potential run-time errors. Consequently, they resorted to including rudimentary security provisions through these blocks. While these `try-catch` statements were effective in preventing certain Denial of Service attacks in some scenarios, they did not significantly improve the overall security of the code in other cases. However, it is noteworthy that for prompting techniques like *zero-shot CoT* and *RCI*, the introduction of `try-catch` blocks was complemented by the integration of additional pertinent security measures, providing a more comprehensive approach to code security.

(iii) *Addition of Unnecessary Security Measures*: Frequently, the models exhibit uncertainty regarding the appropriate security measures to be included in the generated code. This uncertainty becomes particularly noticeable in the context of *naive-secure* prompts, where the specific security requirements are not explicitly evident from the prompt itself. To illustrate this point using our findings, in a coding task where the primary objective is to copy content from a source variable to a target variable, GPT-3.5 directed its attention toward securely hashing the data to be copied from the source variable. This extra step, although a security measure, was unnecessary and not mentioned in the original coding task. This observation suggests the importance of directing the focus of the LLM to the desired security aspect when utilizing zero-shot prompts, as it helps mitigate ambiguity and guides the model toward more relevant and focused security enhancements within the generated code.

(iv) *Additional Validation Checks*: Within the code generated through the utilization of the *RCI* prompting technique, a notable increase in the presence of validity checks is observed especially in code generated using GPT-3.5 and GPT-4. These checks primarily serve the purpose of validating input received from external sources, such as external function calls or user inputs. These checks encompass a wide range of potential error scenarios, including security-related input validations. The *RCI* technique, which encourages the model to enhance its own code based on self-feedback, has resulted in the model's ability to recognize its own shortcomings. Consequently, this has led to a substantial increase in the number of both functional and security-related checks integrated into the generated code.

(v) *Security-Related Comments*: Some code generated by both GPT-3.5 and GPT-4 include warnings highlighting potential vulnerabilities. These warnings are present in code generated using all prompt templates except the *baseline* prompt. While these comments do not directly enhance the code's security, they serve as valuable aids for developers utilizing such models to identify security-related aspects within the code. Notably, code produced through the *RCI* and *zero-shot CoT* methods by GPT-3.5 and GPT-4 stands out for its detailed comments regarding the security measures implemented in the code. Moreover, there are instances in which code snippets generated using all prompting techniques contain additional comments pertaining to how to enhance security, even though these enhancements are not actually implemented. This behavior is observed across all prompt types except the *baseline* one. Specifically, GPT-4-generated code, when prompted with *zero-shot CoT* and *RCI* prompts, often includes a section titled "Additional Security Considerations." In this section, an extensive list of potential security measures that can or should be implemented to further enhance security is provided. Examples of such measures encompass suggestions like "Use a secure database connection like SSL/TSL" and "Ensure script permissions are correctly configured to prevent unauthorized access or modifications." Furthermore, many code snippets generated through the *RCI* method also include cautionary security warnings, such as "Avoid logging sensitive information" and "Ensure that memory dumps do not contain private data." These comments, while not directly affecting the code's functionality, serve as valuable reminders for developers to consider security aspects during the coding process.

(vi) *Calls to Undeclared/Undefined Secure Methods*: We observed numerous cases in GPT-3.5 and GPT-4 where the generated Python snippet included calls to undeclared methods that did not exist within the code's scope. There were also calls to declared methods that remained incomplete. In many cases, these methods are responsible for implementing security-sensitive tasks such as password, or session verification, and are frequently accompanied by security-related comments such as "securely verify the user session." This is mainly observed in zero-shot prompt variants. Our analysis suggests that the models acknowledge the necessary security measures required in the code from the prompts, but have prioritized their efforts on fulfilling the functional requirements specified in the prompt. Code snippets with incomplete logic were removed from our security analysis in the code validity analysis step.

(vii) *Modification of Method Names*: Quite commonly, when employing *zero-shot* prompt variations, we observe a pattern where method names in the generated code are prefixed with the term "secure." For instance, we came across method names like `secure_ping()`, `secure_memory_allocation`, `secure_upload_file`, and the like. However, it is noteworthy that in many cases, the actual implementation within these methods remains unaltered, despite the suggestive "secure" prefixes in the method names. This tendency is particularly prevalent in code generated by GPT-3 and GPT-3.5.

7.5 Generalizability to C

The transferability of the findings obtained for Python regarding the impact of different prompting techniques was tested on C, as mentioned in Section 5.4. The results from the CodeQL evaluations are presented in Table 6. As in the case of Python (Section 6.3), we selected a sample of results (49 cases, which account for 10% of the results) from CodeQL for manual inspection. After conducting a reliability agreement test, we obtained a Kappa value of 0.82 averaged over the results of the seven prompting techniques from GPT-4, suggesting that the results for C code from CodeQL are highly reliable.

It can be seen that all the prompt templates that incorporated some form of security specifications showed improved results compared to the *baseline* prompt (which did not include any security-related instructions). Hence, this further highlights the importance of including explicit security specifications in prompts when generating code using LLMs. Among all the techniques, *RCI* produced C code with the lowest weakness count, rate, and density, consistent with the findings on Python. This, in principle, suggests its applicability across different programming languages. However, unlike the Python results, where the *persona/memetic proxy* technique consistently performed the worst, the *CWE-specific* template using the zero-shot technique performed the worst in C (of course, aside from the *baseline* prompt). Overall, the findings from the C language experiment suggest that while the RCI technique remains the most effective, the relative performance of other prompting techniques may vary depending on the programming language.

Further data from this experiment including the LLM-generated C code files, the CodeQL responses and the type of security weaknesses detected in the code are provided in the replication package.

Actionable Takeaways. When using LLMs or LLM-powered tools like ChatGPT or Copilot, which enable NL user interaction in a software development environment, the following considerations must be taken into account:

(1) Using RCI is preferable over the other techniques studied in this work, as RCI can largely improve the security of the generated code (up to an order of magnitude w.r.t weakness density) even when applied with just two iterations. This technique has stayed valuable over several versions of the LLM models, and, hence, there is an expectation that it will stay valid in the future as well.
(2) As of today, state-of-the-art LLM provides better results in terms of weakness density when used for C generation w.r.t Python, and therefore, it should be used with more caution in the latter case.
(3) Nevertheless, the use of RCI might bring the defect density in Python to the same ballpark as in C. Therefore, this might justify the use of a slightly more complex prompting technique for software development in practice.
(4) In cases where multi-step techniques like RCI are not feasible, using simple zero-shot prompting with templates similar to *comprehensive* prompts, that specify well-established secure coding standards, can provide comparable results in relation to more complex techniques.
(5) Even with the use of secure prompting techniques, coding tasks involving web application development and OS command execution can still result in security weaknesses due to their complex nature. Special attention should be given to such tasks, particularly for vulnerabilities related to CWE-259:*Hard-coded Passwords* (or more broadly CWE-798: *Hard-coded Credentials*), CWE-94 and CWE-78:*OS Command Injection*.
(6) Static analysis tools like Bandit and CodeQL seem to be able to detect the issues that LLMs may overlook (such as CWE-259, CWE-94, CWE-78). Hence, it is still strongly advised to use these tools in tandem with LLMs in the development pipeline.

8 Impact of Data Leakage

Evaluating widely used closed-source LLMs such as the ones used in this study poses the risk of data leakage [123]. Data leakage (also referred to as data contamination) [7] occurs when models

Table 6. The Results of Validity and Security Analysis of C Code Generated by GPT-4 Using the Seven Prompt Templates

GPT-4							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Baseline (0-shot)	67	6	91	27.88	19	0.283	0.009
Naive-secure (0-shot)	67	8	86	35.98	15	0.223	0.005
CWE-specific (0-shot)	67	24	95	41.5	19	0.283	0.006
Comprehensive (0-shot)	67	17	95	50.97	13	0.194	0.004
Zero-shot CoT	67	8	81	36.95	12	0.179	0.004
RCI	67	13	157	56.80	11	0.164	**0.002**
Persona/memetic proxy	67	21	73	40.29	11	0.164	0.004

The *count* is the total number of security weaknesses detected by CodeQL, *rate* is the average number of security weaknesses per code, and *density* is the average number of security weaknesses per LOC. The best performing technique is highlighted in bold text.

have prior exposure to the benchmark datasets used for their evaluation, which can lead to false estimations of their capabilities. Balloccu et al. [7] identified two forms of data leakage: direct and indirect.

Direct data leakage occurs when evaluation data is already included in a model's training data. Considering that the OpenAI models used in this study are closed-source and their training data is undisclosed, this poses a potential concern. The LLMSecEval dataset used in this study, comprising NL code generation prompts paired with secure Python implementations, was formally published in May 2023. However, its corresponding GitHub public repository was created in January 2023. The OpenAI documentation[5] states that GPT-3 (text-davinci-002) and GPT-3.5 (gpt-3.5-turbo) were trained on data up to September 2021, eliminating the risk of direct data leakage for these models. However, GPT-4 (gpt-4-1105-preview) was trained on data up to April 2023, suggesting a slight risk of leakage since the GitHub repository predates this one. To verify the extent of direct data leakage in the code generated by GPT-4, we performed a leakage test on the Python code generated with the RCI technique, the best-performing prompting technique in our experiments. For this, we employed the *Dolos toolkit* [67], which is a source code plagiarism detection tool. Dolos works by tokenizing and canonicalizing programs into Abstract Syntax Tree representations to calculate a similarity score that captures semantic-level similarity through k-gram matching between the source and target programs. This tool has been used in several studies for quantifying contamination in code generated by LLMs [89, 125]. The average similarity score obtained for all the code generated using RCI technique and the secure implementation present in the LLMSecEval dataset is 0.075. Yu et al. [125] in their study shows that a similarity score greater than 0.5 indicates potential plagiarism. Hence, this suggests that the impact of the direct data leakage/contamination in our results is very minimal.

On the other hand, indirect leakage happens when models learn from user interactions via **Reinforcement Learning through Human Feedback (RLHF)**. The OpenAI documentation[6] states that only data from web interface interactions, not from API usage, is utilized for this purpose. Since this study exclusively relied on API interactions, there is no risk of indirect data leakage.

[5] https://platform.openai.com/docs/models.
[6] https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance.

9 Threats to Validity

Although this study yields valuable findings, it is important to acknowledge certain limitations.

Construct Validity. The validity analysis of code responses generated by all LLMs was conducted by a single author, potentially introducing biases in the evaluation process. Nonetheless, efforts were made to mitigate such biases by explicitly outlining the criteria for assessing the validity of code snippets, as detailed in Section 5.2. We also acknowledge that the prompting techniques underwent evaluation using prompt templates created by us. These generated templates might have influenced the results obtained for each technique from the LLMs. However, attention was given to crafting the templates, adhering closely to the design and examples that demonstrated optimal results in the respective papers that introduced these techniques.

External Validity. This study was conducted only using the OpenAI models. As previously stated in 5.1, this decision was made due to the popularity of these models in the prompt engineering literature and their demonstrated proficiency in handling coding tasks articulated in NL, as identified during a preliminary model selection examination by us. It is also worth mentioning that, we focused on prompting techniques that do not rely on demonstrative examples. This choice stemmed from a user study [78] which highlighted that users predominantly interacted with AI assistants using NL coding task specifications or instructions, without supplying demonstrative examples. We also note that the responses generated by the LLMs were not subjected to a randomness check. Since the code validity analysis was conducted manually, generating and evaluating multiple random responses from the LLMs were not feasible. However, we used 150 coding tasks to assess the impact of each prompting technique, which helps smooth out the fluctuation in results caused by randomness to some extent. Furthermore, the LLMSecEval dataset contains NL prompts for only 18 out of *Top 25* CWEs of the year 2021. However, 15 out of the 18 CWEs considered in this study have retained their position on the *Top 25* list of 2022 and 2023 proving the continued relevance and significance of our research findings. Additionally, while our agreement test between manual and Bandit security analyses demonstrates Bandit's reliability to an extent, we acknowledge that it is not perfect. Some specific security weaknesses may have gone undetected. Nevertheless, Bandit proved sufficiently reliable for comparing the relative effectiveness of different prompting techniques.

10 Conclusion

In an era where software development increasingly relies on automatic code generators, it is crucial to ensure the security of the code that LLMs produce out of NL descriptions. Through a literature review, we identified 15 distinct prompting techniques that can be applied to code generation. We also classified these techniques into five categories based on the prompting strategy they follow among other characteristics. Based on the suitability for the secure code generation task, we conducted an in-depth analysis of four prompting techniques to gauge their impact on secure code generation using GPT-3, GPT-3.5, and GPT-4.

Our analysis reaffirms the prevalence of security weaknesses in code generated by LLMs when prompted with NL instructions, with significant challenges stemming from CWE-78, CWE-259, CWE-94, and CWE-330. Among the prompting techniques investigated, *RCI*, a refinement-based approach, exhibited notable effectiveness in preventing security weaknesses in LLM-generated code. Particularly noteworthy was its performance with GPT-4, where it reduced the average weakness density by 77.5% compared to baseline prompting that includes no security specifications. Although RCI demonstrated the highest performance, to the extent of our knowledge, this technique has not been applied for secure code generation in existing literature. This highlights the need for additional research to investigate refinement-based methods, like RCI, which leverage self-critiquing and improvement capabilities of LLMs to enhance security in LLM-generated code. *Zero-shot* prompting

yielded surprisingly favorable outcomes considering its straightforward nature, performing better than *zero-shot CoT* and *persona/memetic proxy* yet falling short of *RCI*. However, *zero-shot* prompting holds promise due to its simplicity and relative performance, provided an optimal prompt can be identified.

Notably, recent advancements in prompt optimization techniques, such as genetic algorithms and black-box tuning, offer avenues for automatically optimizing prompts for various text-generation tasks. Future work can focus on exploring these optimization approaches to identify the optimal prompts for *RCI* and *zero-shot* techniques for secure code generation.

11 Replication Package

All data collected and generated in this study are available in https://figshare.com/s/195a75d8c4dd86816223. This repository contains the results of the literature review along with the information on the prompting techniques that were removed from our final selection and the rationale behind this exclusion. Furthermore, the repository contains the code generated by all three LLMs for the seven prompt templates, including their validity and security analysis results.

References

[1] MITRE. 2024. MITRE. Retrieved October 10, 2023 from https://www.mitre.org/

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '21)*. Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.), Association for Computational Linguistics, 2655–2668. DOI: https://doi.org/10.18653/V1/2021.NAACL-MAIN.211

[3] Simran Arora, Avanika Narayan, Mayee F. Chen, Laurel J. Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Ré. 2023. Ask Me Anything: A simple strategy for prompting language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=bhUPJnS2g0X

[4] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129. DOI: https://doi.org/10.1007/S10664-023-10380-1

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[6] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional AI: Harmlessness from AI feedback. arXiv:2212.08073. Retrieved from https://arxiv.org/abs/2212.08073

[7] Simone Balloccu, Patrícia Schmidtová, Mateusz Lango, and Ondrej Dusek. 2024. Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source LLMs. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics—Volume 1: Long Papers (EACL '24)*. Yvette Graham and Matthew Purver (Eds.), Association for Computational Linguistics, 67–93. Retrieved from https://aclanthology.org/2024.eacl-long.5

[8] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)* 57, 1 (1995), 289–300.

[9] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP '13)*. A meeting of SIGDAT, a Special Interest Group of the ACL. ACL, 1533–1544. Retrieved from https://aclanthology.org/D13-1160/

[10] Sumithra Bhakthavatsalam, Daniel Khashabi, Tushar Khot, Bhavana Dalvi Mishra, Kyle Richardson, Ashish Sabharwal, Carissa Schoenick, Oyvind Tafjord, and Peter Clark. 2021. Think you have solved direct-answer question answering? Try ARC-DA, the direct-answer AI2 reasoning challenge. arXiv:2102.03315. Retrieved from https://arxiv.org/abs/2102.03315

[11] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20), the*

32nd Innovative Applications of Artificial Intelligence Conference (IAAI '20), the 10th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI '20). AAAI Press, 7432–7439. DOI: https://doi.org/10.1609/AAAI.V34I05.6239

[12] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. GPT-NeoX-20B: An open-source autoregressive language model. arXiv:2204.06745. Retrieved from https://arxiv.org/abs/2204.06745

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020 (NeurIPS '20)*, Virtual. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[14] Angela Carrera-Rivera, William Ochoa, Felix Larrinaga, and Ganix Lasa. 2022. How-to conduct a systematic literature review: A quick guide for computer science research. *MethodsX* 9 (Nov. 2022), 101895. DOI: https://doi.org/10.1016/j.mex.2022.101895

[15] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691. DOI: https://doi.org/10.1109/TSE.2023.3267446

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[17] Eunsol Choi, He He, Mohit Iyyer, Mark Yatskar, Wen-tau Yih, Yejin Choi, Percy Liang, and Luke Zettlemoyer. 2018. QuAC: Question answering in context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.), Association for Computational Linguistics, 2174–2184. DOI: https://doi.org/10.18653/V1/D18-1241

[18] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24 (2023), 240:1–240:113. Retrieved from http://jmlr.org/papers/v24/22-1144.html

[19] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv:2110.14168. Retrieved from https://arxiv.org/abs/2110.14168

[20] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P. Xing, and Zhiting Hu. 2022. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP '22)*. Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.), Association for Computational Linguistics, 3369–3391. DOI: https://doi.org/10.18653/V1/2022.EMNLP-MAIN.222

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (NAACL-HLT '19)*. Jill Burstein, Christy Doran, and Thamar Solorio (Eds.), Association for Computational Linguistics, 4171–4186. DOI: https://doi.org/10.18653/V1/N19-1423

[22] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating code generation by learning code execution. arXiv:2301.09043. Retrieved from https://arxiv.org/abs/2301.09043

[23] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (NAACL-HLT '19)*. Jill Burstein, Christy Doran, and Thamar Solorio (Eds.), Association for Computational Linguistics, 2368–2378. DOI: https://doi.org/10.18653/V1/N19-1246

[24] Olive JeanDunn. 1961. Multiple comparisons among means. *Journal of the American Statistical Association* 56, 293 (1961), 52–64. Retrieved from http://www.jstor.org/stable/2282330

[25] Nadir Durrani, Barry Haddow, Philipp Koehn, and Kenneth Heafield. 2014. Edinburgh's phrase-based machine translation systems for WMT-14. In *Proceedings of the 9th Workshop on Statistical Machine Translation (WMT@ACL '14)*. Association for Computer Linguistics, 97–104. DOI: https://doi.org/10.3115/V1/W14-3309

[26] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing. arXiv:2104.02443. Retrieved from https://arxiv.org/abs/2104.02443

[27] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. arXiv:2310.03533. Retrieved from https://arxiv.org/abs/2310.03533

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event*. Trevor Cohn, Yulan He, and Yang Liu (Eds.), Association for Computational Linguistics, 1536–1547.

[29] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=hQwb-lbM6EL

[30] Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=yf1icZHC-l9

[31] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamile Lukosiute, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, et al.. 2023. The capacity for moral self-correction in large language models. arXiv:2302.07459. Retrieved from https://arxiv.org/abs/2302.07459

[32] Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. 2021. Did Aristotle use a laptop? A question answering benchmark with implicit reasoning strategies. *Transactions of the Association for Computational Linguistics* 9 (2021), 346–361. DOI: https://doi.org/10.1162/TACL_A_00370

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR '21)*, Virtual Event. OpenReview.net. Retrieved from https://openreview.net/forum?id=jLoC4ez43PZ

[34] Chengcheng Han, Liqing Cui, Renyu Zhu, Jianing Wang, Nuo Chen, Qiushi Sun, Xiang Li, and Ming Gao. 2023. When gradient descent meets derivative-free optimization: A match made in black-box scenario. In *Findings of the Association for Computational Linguistics: ACL 2023*. Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.), Association for Computational Linguistics, 868–880. DOI: https://doi.org/10.18653/V1/2023.FINDINGS-ACL.55

[35] Anne-Wil Harzing. 2016. Publish or Perish. Retrieved from https://harzing.com/resources/publish-or-perish

[36] Mohammadreza Hazhirpasand, Mohammad Ghafari, Stefan Krüger, Eric Bodden, and Oscar Nierstrasz. 2019. The impact of developer experience in using java cryptography. In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*. IEEE, 1–6. DOI: https://doi.org/10.1109/ESEM.2019.8870184

[37] Jingxuan He and Martin T. Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.), ACM, 1865–1879. DOI: https://doi.org/10.1145/3576915.3623175

[38] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks'21)*, Virtual. Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Retrieved from https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[39] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks '21)*, Virtual. Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Retrieved from https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html

[40] Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. 2014. Learning to solve arithmetic word problems with verb categorization. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14), A meeting of SIGDAT, a Special Interest Group of the ACL*. Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.), ACL, 523–533. DOI: https://doi.org/10.3115/V1/D14-1058

[41] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*. Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.), Association for Computational Linguistics, 1049–1065. DOI: https://doi.org/10.18653/V1/2023.FINDINGS-ACL.67

[42] Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, et al. 2022. Do as I can, not as I say: Grounding language in robotic affordances. In

Proceedings of the Conference on Robot Learning (CoRL '2). Karen Liu, Dana Kulic, and Jeffrey Ichnowski (Eds.), Proceedings of Machine Learning Research, Vol. 205, PMLR, 287–318. Retrieved from https://proceedings.mlr.press/v205/ichter23a.html

[43] GitHub Inc. 2025. CodeQL. Retrieved January 9, 2024 from https://codeql.github.com/docs/

[44] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1219–1231.

[45] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP '21)*, Virtual Event. Marie-FrancineMoens, XuanjingHuang, LuciaSpecia, and Scott Wen-tauYih (Eds.), Association for Computational Linguistics, 5954–5971. DOI: https://doi.org/10.18653/V1/2021.EMNLP-MAIN.482

[46] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large language models and simple, stupid bugs. In *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR '23)*. IEEE, 563–575. DOI: https://doi.org/10.1109/MSR59073.2023.00082

[47] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning code generation with large language models. arXiv:2303.06689. Retrieved from https://arxiv.org/abs/2303.06689

[48] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. 2017. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers (ACL '17)*. Regina Barzilay and Min-Yen Kan (Eds.), Association for Computational Linguistics, 1601–1611. DOI: https://doi.org/10.18653/V1/P17-1147

[49] Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. 2020. UnifiedQA: Crossing format boundaries with a single QA system. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event*. Trevor Cohn, Yulan He, and Yang Liu (Eds.), Association for Computational Linguistics, 1896–1907. DOI: https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.171

[50] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. In *Proceedings of the Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023 (NeurIPS '23)*. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2023/hash/7cc1005ec73cfbaac9fa21192b622507-Abstract-Conference.html

[51] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022 (NeurIPS '22)*. Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[52] Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. 2015. Parsing algebraic word problems into equations. *Transactions of the Association for Computational Linguistics* 3 (2015), 585–597. DOI: https://doi.org/10.1162/TACL_A_00160

[53] William H. Kruskal and W. Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association* 47, 260 (1952), 583–621. DOI: https://doi.org/10.1080/01621459.1952.10483441

[54] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 452–466. DOI: https://doi.org/10.1162/TACL_A_00276

[55] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard H. Hovy. 2017. RACE: Large-scale ReAding comprehension dataset from examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP '17)*. Martha Palmer, Rebecca Hwa, and Sebastian Riedel (Eds.), Association for Computational Linguistics, 785–794. DOI: https://doi.org/10.18653/V1/D17-1082

[56] Brenden M. Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML '18)*. Jennifer G. Dy and Andreas Krause (Eds.), Proceedings of Machine Learning Research, Vol. 80, PMLR, 2879–2888. Retrieved from http://proceedings.mlr.press/v80/lake18a.html

[57] Andrew K. Lampinen, Ishita Dasgupta, Stephanie C. Y. Chan, Kory W. Mathewson, Michael Henry Tessler, Antonia Creswell, James L. McClelland, Jane Wang, and Felix Hill. 2022. Can language models learn from explanations in context? In *Findings of the Association for Computational Linguistics: EMNLP 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.), Association for Computational Linguistics, 537–563. DOI: https://doi.org/10.18653/V1/2022.FINDINGS-EMNLP.38

[58] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Proceedings of the Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022 (NeurIPS '22)*. Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html

[59] Leandro von Werra Lewis Tunstall and Thomas Wolf. 2022. *Natural Language Processing with Transformers*. O'Reilly Media, Inc. (2022).

[60] Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. 2021. *Jurassic-1: Technical Details and Evaluation*. AI21 Labs Technical Report. Retrieved from https://uploads-ssl.webflow.com/60fd4503684b466578c0d307/61138924626a6981ee09caf6_jurassic_tech_paper.pdf

[61] Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. 2020. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2022)*. Trevor Cohn, Yulan He, and Yang Liu (Eds.), Association for Computational Linguistics, 1823–1840. DOI: https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.165

[62] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers (ACL '17)*. Regina Barzilay and Min-Yen Kan (Eds.), Association for Computational Linguistics, 158–167. DOI: https://doi.org/10.18653/V1/P17-1015

[63] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Proceedings of the Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023 (NeurIPS '23)*. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[64] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks '21)*, Virtual. Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Retrieved from https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[65] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. arXiv:2302.07867. Retrieved from https://arxiv.org/abs/2302.07867

[66] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Proceedings of the Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023 (NeurIPS '23)*. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html

[67] Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. 2022. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning* 38, 4 (2022), 1046–1061. DOI: https://doi.org/10.1111/JCAL.12662

[68] Mitchell P. Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*. Morgan Kaufmann. Retrieved from https://aclanthology.org/H94-1020/

[69] Shikib Mehri and Maxine Eskénazi. 2020. Unsupervised evaluation of interactive dialog with DialoGPT. In *Proceedings of the 21st Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGdial '20) 1st Virtual Meeting*. Olivier Pietquin, Smaranda Muresan, Vivian Chen, Casey Kennington, David Vandyke, Nina Dethlefs, Koji Inoue, Erik Ekstedt, and Stefan Ultes (Eds.), Association for Computational Linguistics, 225–235. Retrieved from https://aclanthology.org/2020.sigdial-1.28/

[70] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.), Association for Computational Linguistics, 2381–2391. DOI: https://doi.org/10.18653/V1/D18-1260

[71] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James F. Allen. 2016. A corpus and evaluation framework for deeper understanding of commonsense stories. arXiv:1604.01696. Retrieved from http://arxiv.org/abs/1604.01696

[72] Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2020. Adversarial NLI: A new benchmark for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association*

for Computational Linguistics (ACL '20). Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.), Association for Computational Linguistics, 4885–4901. DOI: https://doi.org/10.18653/V1/2020.ACL-MAIN.441

[73] OpenAI. 2023. GPT-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[74] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers (ACL '16)*. Association for Computer Linguistics. DOI: https://doi.org/10.18653/V1/P16-1144

[75] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '21)*. Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.), Association for Computational Linguistics, 2080–2094. DOI: https://doi.org/10.18653/V1/2021.NAACL-MAIN.168

[76] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '22)*. IEEE, 754–768. DOI: https://doi.org/10.1109/SP46214.2022.9833571

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

[78] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. WeizhiMeng, Christian Damsgaard Jensen, CasCremers, and EnginKirda (Eds.), ACM, 2785–2799. DOI: https://doi.org/10.1145/3576915.3623157

[79] Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task learning with code-text transformer. arXiv:2105.08645. Retrieved from https://arxiv.org/abs/2105.08645

[80] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. 2023. GrIPS: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics (EACL '23)*. Andreas Vlachos and Isabelle Augenstein (Eds.), Association for Computational Linguistics, 3827–3846. DOI: https://doi.org/10.18653/V1/2023.EACL-MAIN.277

[81] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, et al. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks '21)*, Virtual. Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Retrieved from https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html

[82] PyCQA. 2025. Bandit Documentation. Retrieved January 5, 2024 from https://bandit.readthedocs.io/en/latest/index.html

[83] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. Retrieved from https://api.semanticscholar.org/CorpusID:160025533

[84] Md. Rayhanur Rahman, Akond Rahman, and Laurie A. Williams. 2019. Share, but be aware: Security smells in Python Gists. In *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME '19)*. IEEE, 536–540. DOI: https://doi.org/10.1109/ICSME.2019.00087

[85] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Volume 2: Short Papers (ACL '18*). Iryna Gurevych and Yusuke Miyao (Eds.), Association for Computational Linguistics, 784–789. DOI: https://doi.org/10.18653/V1/P18-2124

[86] Irum Rauf, Marian Petre, Thein Tun, Tamara Lopez, Paul Lunn, Dirk van der Linden, John N. Towse, Helen Sharp, Mark Levine, Awais Rashid, et al. 2022. The case for adaptive security interventions. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2022), 9:1–9:52. DOI: https://doi.org/10.1145/3471930

[87] Siva Reddy, Danqi Chen, and Christopher D. Manning. 2019. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics* 7 (2019), 249–266. DOI: https://doi.org/10.1162/TACL_A_00266

[88] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In*: Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '21), Virtual Event*, Extended Abstracts, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, and Takeo Igarashi (Eds.), ACM, 314:1–314:7. DOI: https://doi.org/10.1145/3411763.3451760

[89] Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers) (ACL '24). Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, 14116–14137. DOI: https://doi.org/10.18653/V1/2024.ACL-LONG.761

[90] Subhro Roy and Dan Roth. 2015. Solving general arithmetic word problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP '15)*. Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton (Eds.), The Association for Computational Linguistics, 1743–1752. DOI: https://doi.org/10.18653/V1/D15-1202

[91] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. 2021. A large-scale security-oriented static analysis of Python packages in PyPI. In *Proceedings of the 18th International Conference on Privacy, Security and Trust (PST '21)*. IEEE, 1–10. DOI: https://doi.org/10.1109/PST52912.2021.9647791

[92] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. WinoGrande: An adversarial Winograd schema challenge at scale. *Communications of the ACM* 64, 9 (2021), 99–106. DOI: https://doi.org/10.1145/3474381

[93] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A user study on the security implications of large language model code assistants. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23)*. Joseph A. Calandrino and Carmela Troncoso (Eds.), USENIX Association, 2205–2222. Retrieved from https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

[94] Advait Sarkar, Carina Negreanu, Ben Zorn, Sruti Srinivasa Ragavan, Christian Pölitz, and Andrew D. Gordon. 2022. What is it like to program with artificial intelligence? In *Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG '22)*. Simon Holland, Marian Petre, Luke Church, and Mariana Marasoiu (Eds.), Psychology of Programming Interest Group, 127–153. Retrieved from https://ppig.org/papers/2022-ppig-33rd-sarkar/

[95] William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. arXiv:2206.05802. Retrieved from https://arxiv.org/abs/2206.05802

[96] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna C. S. Santos. 2024. Quality assessment of ChatGPT generated code and their use by developers. In *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR '24)*. Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou (Eds.), ACM, 152–156. DOI: https://doi.org/10.1145/3643991.3645071

[97] Mohammed Latif Siddiq and Joanna C. S. Santos. 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 (MSR4P&S '22)*. ACM, New York, NY, 29–33. DOI: https://doi.org/10.1145/3549035.3561184

[98] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. arXiv:2206.04615. Retrieved from https://arxiv.org/abs/2206.04615

[99] Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. 2022. Black-box tuning for language-model-as-a-service. In *Proceedings of the International Conference on Machine Learning (ICML '22)*. Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (Eds.), Proceedings of Machine Learning Research, Vol. 162, PMLR, 20841–20855. Retrieved from https://proceedings.mlr.press/v162/sun22e.html

[100] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, et al. 2023. Challenging BIG-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*. Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.), Association for Computational Linguistics, 13003–13051. DOI: https://doi.org/10.18653/V1/2023.FINDINGS-ACL.824

[101] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. CommonsenseQA: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (NAACL-HLT '19)*. Jill Burstein, Christy Doran, and Thamar Solorio (Eds.), Association for Computational Linguistics, 4149–4158. DOI: https://doi.org/10.18653/V1/N19-1421

[102] James Thomas and Angela Harden. 2008. Methods for the thematic synthesis of qualitative research in systematic reviews. *BMC Medical Research Methodology* 8 (2008), 45.

[103] Catherine Tony, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. 2022. GitHub considered harmful? Analyzing open-source projects for the automatic generation of cryptographic API call sequences. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability and Security (QRS '22)*. IEEE, 896–906. DOI: https://doi.org/10.1109/QRS57517.2022.00094

[104] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A dataset of natural language prompts for security evaluations. In *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR '23)*. IEEE, 588–592. DOI: https://doi.org/10.1109/MSR59073.2023.00084

[105] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971

[106] Peter D. Turney, Michael L. Littman, Jeffrey Bigham, and Victor Shnayder. 2003. Combining independent modules to solve multiple-choice synonym and analogy problems. arXiv:cs/0309035. Retrieved from https://arxiv.org/abs/cs/0309035

[107] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '22), Extended Abstracts*. Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.), ACM, 332:1–332:7. DOI: https://doi.org/10.1145/3491101.3519665

[108] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. In *Proceedings of the Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019 (NeurIPS '19)*. HannaM. Wallach, HugoLarochelle, AlinaBeygelzimer, Florenced'Alché-Buc, EmilyB. Fox, and RomanGarnett (Eds.), 3261–3275. Retrieved from https://proceedings.neurips.cc/paper/2019/hash/4496bf24afe7fab6f046bf4923da8de6-Abstract.html

[109] Boshi Wang, Xiang Deng, and Huan Sun. 2022. Iteratively prompt pre-trained language models for chain of thought. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP '22)*. Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.), Association for Computational Linguistics, 2714–2730. Retrieved from https://aclanthology.org/2022.emnlp-main.174

[110] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=1PL1NIMMrw

[111] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware1 unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP '21)*, Virtual Event. Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.), Association for Computational Linguistics, 8696–8708. DOI: https://doi.org/10.18653/v1/2021.emnlp-main.685

[112] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22)*. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[113] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv:2302.11382. Retrieved from https://arxiv.org/abs/2302.11382

[114] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. arXiv:2303.07839. Retrieved from https://arxiv.org/abs/2303.07839

[115] Anna-Katharina Wickert, Lars Baumgärtner, Florian Breitfelder, and Mira Mezini. 2021. Python crypto misuses in the wild. In*: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21)*. Filippo Lanubile, Marcos Kalinowski, and Maria Teresa Baldassarre (Eds.), ACM, 31:1–31:6. DOI: https://doi.org/10.1145/3475716.3484195

[116] Anna-Katharina Wickert, Michael Reif, Michael Eichberg, Anam Dodhy, and Mira Mezini. 2019. A dataset of parametric cryptographic misuses. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.), IEEE/ACM, 96–100. DOI: https://doi.org/10.1109/MSR.2019.00023

[117] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)*. Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit (Eds.), ACM, 38:1–38:10. DOI: https://doi.org/10.1145/2601248.2601268

[118] Yu Xiao and Maria Watson. 2019. Guidance on conducting a systematic literature review. *Journal of Planning Education and Research* 39, 1 (2019), 93–112. DOI: https://doi.org/10.1177/0739456X17723971

[119] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022 A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS@PLDI '22)*. Swarat Chaudhuri and Charles Sutton (Eds.), ACM, 1–10. DOI: https://doi.org/10.1145/3520312.3534862

[120] Hanwei Xu, Yujun Chen, Yulun Du, Nan Shao, Yanggang Wang, Haiyu Li, and Zhilin Yang. 2022. GPS: Genetic prompt search for efficient few-shot learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP '22)*. Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.), Association for Computational Linguistics, 8162–8171. DOI: https://doi.org/10.18653/V1/2022.EMNLP-MAIN.559

[121] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Proceedings of the Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023 (NeurIPS '23)*. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html

[122] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Eric Sun, and Yue Zhang. 2023. A survey on Large Language Model (LLM) security and privacy: The good, the bad, and the ugly. arXiv:2312.02003. Retrieved from https://arxiv.org/abs/2312.02003

[123] He Ye, Zimin Chen, and Claire Le Goues. 2023. PreciseBugCollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 1899–1910. DOI: https://doi.org/10.1109/ASE56229.2023.00163

[124] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778. Retrieved from https://arxiv.org/abs/2304.10778

[125] Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. 2023. CodeIP-Prompt: Intellectual property infringement assessment of code language models. In *Proceedings of the International Conference on Machine Learning (ICML '23)*. Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Proceedings of Machine Learning Research, Vol. 202, PMLR, 40373–40389. Retrieved from https://proceedings.mlr.press/v202/yu23g.html

[126] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Conference of the Association for Computational Linguistics, Volume 1: Long Papers (ACL '19)*. Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.), Association for Computational Linguistics, 4791–4800. DOI: https://doi.org/10.18653/V1/P19-1472

[127] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In*: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), Virtual Event*. Sukyoung Ryu and Yannis Smaragdakis (Eds.), ACM, 39–51. DOI: https://doi.org/10.1145/3533767.3534390

[128] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. *In Proceedings of the Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*. Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.), 649–657. Retrieved from https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html

[129] Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. 2023. Progressive-hint prompting improves reasoning in large language models. arXiv:2304.09797. Retrieved from https://arxiv.org/abs/2304.09797

[130] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X. arXiv:2303.17568. Retrieved from https://arxiv.org/abs/2303.17568

[131] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, et al. 2023. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=WZH7099tgfM

[132] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*). OpenReview.net. Retrieved from https://openreview.net/pdf?id=92gvk82DE-

Appendices

A Manual Verification of SAST Tool Results

Below, we detail the full process followed to manually verify the security analysis results of the SAST tools (Bandit and CodeQL).

A.1 Step 1: Manual Security Weakness Detection

We began by manually reviewing the code snippets (for the randomly selected tasks) independently of the SAST tool results to identify any of the 18 security weaknesses documented in LLMSecEval. This process involved consulting the official CWE pages on the MITRE Web site, specifically examining the provided potential mitigations and examples of vulnerable code for each weakness. A code snippet was flagged as vulnerable if it lacked the recommended mitigation techniques to prevent the vulnerability or closely resembled the examples of vulnerable code. Consider the example of *CWE-78: OS Command Injection*. One of the mitigation techniques outlined on the CWE page is as follows[7]:

> *Output Encoding*: *While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step.*

Following this mitigation technique, we review the code for any instances of dynamically generated strings, code, or commands that lack validation and encoding. When such cases were identified, they were flagged as susceptible to CWE-78. The example snippet in Listing A1 shows a command that is executed with user-provided information without any validation or encoding, thus leading to CWE-78. Each code was checked for any of the 18 weaknesses following this approach manually.

```
import subprocess
import sys

def get_user_info(username):

        command = f"cat /etc/passwd | grep {username}"
        result = subprocess.run(command, shell=True, capture_output=True, text=True
    ↪ )

        if result.returncode == 0 and result.stdout:
            print(f"User information:\n{result.stdout.strip()}")
        else:
            print(f"No user information found for: {username}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <username>")
    else:
        username = sys.argv[1]
        get_user_info(username)
```

Listing A1. An example Python implementation to demonstrate CWE-78. The code takes a user input to construct a command to be executed.

[7] https://cwe.mitre.org/data/definitions/78.html.

A.2 Step 2: Verification of the Additional CWEs Reported by the SAST Tools

While running Bandit and CodeQL, we did not restrict the scope to the 18 weaknesses in LLM-SecEval. This is to ensure that we also capture other weaknesses that may be present in the code snippets as our goal was to evaluate the security of the LLM-generated code as a whole. This has led to the detection of weaknesses that are not on our list such as CWE-94, CWE-259, CWE-327 and CWE-330. Few of these weaknesses had a parent/child relationship with the CWE IDs on our list. For instance, CWE-259[8] is a child of CWE-798.[9] These were identified as the same weakness in both manual and SAST tool analysis results. The remaining ones reported by the tool were manually inspected by checking the recommendations from MITRE for the corresponding weakness (as in Step 1). The verified weaknesses were added to manual inspection's results.

A.3 Step 3: Agreement Test

The final outcomes of the manual security analysis from Step 2 and the SAST tool results were subjected to a reliability agreement test to evaluate the consistency between the two. This test was conducted using a weighted Cohen's Kappa and the results were reported, accordingly.

B Security Analysis Using CodeQL

As mentioned in Sections 5.3 and 6.3, we also performed security evaluations of the LLM-generated Python code using CodeQL (in addition to Bandit). Table B1 shows the results of the CodeQL evaluation. The table follows the same structure as that of Table 3. Compared to Bandit, CodeQL identified fewer weaknesses in LLM-generated code. Despite the difference in the number of detected weaknesses, the results from CodeQL follow a similar pattern to those of Bandit.

Best Performer. Also according to CodeQL, the RCI prompting technique consistently produces the lowest weakness rate and density in code generated by GPT-3.5 and GPT-4, highlighting its superiority over other techniques.

Worst Performer. Aside from the *baseline* prompting template, the *persona/memetic proxy* technique results in the highest weakness rate in code generated by both GPT-3.5 and GPT-4, confirming the findings from Bandit.

The complete evaluation results generated by CodeQL are provided in the replication package.

[8] https://cwe.mitre.org/data/definitions/259.html.
[9] https://cwe.mitre.org/data/definitions/798.html.

Table B1. The Results of Validity and Security Analysis of Python Code Generated by the Three LLMs Using the Seven Prompt Templates

GPT-3							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Baseline (0-shot)	131	2	80	11.175	53	0.404	0.024
Naive-secure (0-shot)	123	2	31	10.691	19	**0.153**	0.013
CWE-specific (0-shot)	124	3	65	13.846	22	0.177	**0.009**
Comprehensive (0-shot)	120	4	56	15.991	27	0.225	0.015
Zero-shot CoT	126	3	32	10.753	32	0.253	0.021
RCI	125	2	84	20.960	28	0.224	0.012
Persona/memetic proxy	137	5	76	15.875	31	0.226	0.014
GPT-3.5							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Basic (0-shot)	145	3	38	13.889	67	0.462	0.028
Naive-secure (0-shot)	147	3	55	16.374	49	0.333	0.020
CWE-specific (0-shot)	139	3	58	18.733	54	0.038	0.020
Comprehensive (0-shot)	141	5	65	20.680	58	0.375	0.023
Zero-shot CoT	140	3	42	14.357	47	0.335	0.018
RCI	138	5	65	23.543	**35**	**0.253**	**0.008**
Persona/memetic proxy	141	2	42	12.970	57	0.404	0.042
GPT-4							
Prompt Type	**# Valid Code**	**# LOC**			**Security Weaknesses**		
		MIN	**MAX**	**Avg.**	**Count**	**Rate**	**Density**
Basic (0-shot)	144	3	39	16.990	91	0.631	0.032
Naive-secure (0-shot)	149	5	65	21.738	77	0.516	0.020
CWE-specific (0-shot)	145	6	81	28.379	68	0.468	0.014
Comprehensive (0-shot)	147	3	66	26.891	64	0.435	0.013
Zero-shot CoT	146	3	68	22.246	51	0.349	0.016
RCI	143	3	94	39.902	**48**	**0.335**	**0.006**
Persona/memetic proxy	147	3	50	19.319	77	0.523	0.024

The *count* is the total number of security weaknesses detected by CodeQL, *rate* is the average number of security weaknesses per code, and *density* is the average number of security weaknesses per LOC.

Received 4 May 2024; revised 21 January 2025; accepted 20 February 2025