No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT

Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang

Abstract—Large language models (LLMs) have demonstrated impressive capabilities across various natural language processing (NLP) tasks, such as machine translation, question answering, summarization, and so on. Additionally, LLMs are also highly valuable in supporting software engineering tasks, particularly in the field of code generation. Automatic code generation is a process of automatically generating source code or executable code based on given specifications or requirements, improving developer productivity. In this study, we perform a systematic empirical assessment to the quality of code generation using *ChatGPT*, a recent state-of-the-art product LLM. We leverage 728 algorithm problems in five languages (i.e., C, C++, Java, Python, and JavaScript) and 18 CWEs with 54 code scenarios for the code generation task. Our evaluation encompasses a comprehensive analysis of code snippets generated by *ChatGPT*, focusing on three critical aspects: correctness, complexity, and security. We also specifically investigate *ChatGPT*'s ability to engage in multi-round fixing process (i.e., *ChatGPT*'s dialog ability, chatting between users and *ChatGPT* for fixing generated buggy code) of facilitating code generation. By delving into the generated code and examining the experimental results, this work provides valuable insights into the performance of *ChatGPT* in tackling code generation tasks over the three critical aspects. The experimental results demonstrate that (1) *ChatGPT* is better at generating functionally correct code for problems before 2021 in different languages than problems after 2021 with 48.14% advantage in *Accepted* rate on judgment platform, but *ChatGPT*'s ability to directly fix erroneous code with multi-round fixing process to achieve correct functionality is relatively weak; (2) the distribution of cyclomatic and cognitive complexity levels for code snippets in different languages varies. Furthermore, the multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets; (3) in algorithm scenarios with languages of C, C++, and Jave, and CWE scenarios with languages of C and Python3, the code generated by *ChatGPT* has relevant vulnerabilities. However, the multi-round fixing process for vulnerable code snippets demonstrates promising results, with more than 89% of vulnerabilities successfully addressed; and (4) code generation may be affected by *ChatGPT*'s non-determinism factor, resulting in variations of code snippets in functional correctness, complexity, and security. Overall, our findings uncover potential issues and limitations that arise in the *ChatGPT*-based code generation and lay the groundwork for improving AI and LLM-based code generation techniques.

Index Terms—Large Language Model, *ChatGPT*, Code Generation.

1 Introduction

Automatic code generation is a process of automatically generating source code or executable code based on given specifications or requirements. It supports a range of capabilities that benefit software development greatly. By using automatic code generation, developers are able to enhance productivity, reduce development time, and assign more focus to higher-level tasks and core logic. Lots of studies on code generation leverage AI-based approaches [1], [2], [3], [4], [5], especially for using large language models (LLMs) [6], [7], [8], [9], [10], [11] such as the recent *ChatGPT* [12].

AI-based Code Generation. The emergence of AI-based code generation is driven by the increasing complexity of software systems and the desire for a more efficient development process [13]. Traditional code generation approaches [14] rely on predefined templates or rules (e.g., context-free grammar) and input-output specifications, which limits their flexibility and requires manual effort. AI-based approaches [15], [9], [11], [12] leverage the power of machine learning (deep learning) and natural language processing (NLP) to overcome these limitations and can offer more intelligent and adaptable code-generation capabilities. These approaches analyze directly input specifications or requirements expressed in natural language and generate corresponding code snippets or complete programs based on the provided input.

Large Language Model and *ChatGPT*. Recently, large language models (LLMs) demonstrate remarkable capabilities in a wide range of NLP tasks, such as machine translation, question answering, summarization, text generation, grammar checking, and so on [16], [17], [18], [19]. These models possess a capacity for understanding and generating human-like text, approaching the level of humans. LLMs are primarily built on the Transformer architecture [6], with OpenAI's *GPT-3* (Generative Pretrained Transformer 3) [20] being a prominent example. *GPT-3* is trained on extensive amounts of textual data, resulting in exceptional performance. *ChatGPT* [12] is an implementation with dialog ability that is built upon the foundation of *GPT-3.5* [21] (or

Zhijie Liu is with ShanghaiTech University, Shanghai 201210, China. E-mail: liuzhj2022@shanghaitech.edu.cn.
Yutian Tang is with University of Glasgow, United Kingdom. E-mail: yutian.tang@glasgow.ac.uk.
Xiapu Luo is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong SAR, China. E-mail: csxluo@comp.polyu.edu.hk.
Yuming Zhou is with Nanjing University, China. E-mail: zhouyuming@nju.edu.cn.
Liang Feng Zhang is with ShanghaiTech University, Shanghai 201210, China. E-mail: zhanglf@shanghaitech.edu.cn.
Yutian Tang (yutian.tang@glasgow.ac.uk) is the corresponding author.

GPT-4 [22]). It exhibits outstanding performance in areas such as machine translation, question answering, summarization, and so on, and is found in widespread usage in various daily activities. Importantly, *ChatGPT* also possesses the capability of code-related tasks, which can further expand its potential applications. *ChatGPT* now has become an essential tool for individuals, academia, and industry, significantly enhancing productivity in various domains.

Motivation. While AI-based code generation, using LLMs, provides promising advantages in enhancing productivity and automating software development tasks, it is still essential to assess the generated code for showing better insights and understanding. Code generation by LLMs is facing challenges. For example, whether the code generated by LLMs is functionally correct, complex, and secure. The training datasets for LLMs come from the internet, but the quality of the data is uncertain. Subsequently, the quality of the code generated by LLMs also cannot be guaranteed [23], [24], [25], [26]. A deep analysis of these aspects can provide a more comprehensive understanding of AI and LLM-based code generation. In this paper, we are interested in deeply and systematically evaluating the code generated by LLMs in terms of its correctness, complexity, and security. Specifically, we leverage the state-of-the-art *ChatGPT*[1], a recent product, as the representative of LLMs for evaluation, due to its advanced capabilities and widespread recognition [12]. We also assess *ChatGPT*'s dialog ability (i.e., the multi-round fixing process in one single conversation, chatting between users and *ChatGPT* for fixing generated buggy code) in the code generation task over correctness, complexity, and security. By conducting a comprehensive analysis, we seek to uncover potential issues and limitations that arise in the *ChatGPT*-based code generation for improving AI and LLM-based code generation techniques.

Our Study. To cope with the aforementioned challenges and explore the ability of *ChatGPT* [12] to generate code, we collect and leverage 728 algorithm problems in five languages (i.e., C, C++, Java, Python, and JavaScript) and 18 CWEs with 54 code scenarios from *LeetCode* platform [27] and [23], respectively, for the code generation task and intend to answer the following research questions (RQs):

• **RQ1 (Functionally Correct Code Generation):** Is the code generated by *ChatGPT* functionally correct?

• **RQ2 (Multi-round Fixing for Code Generation):** How effective is the multi-round fixing process in improving code generation for functional correctness?

• **RQ3 (Code Complexity):** How complex is the code generated by *ChatGPT*?

• **RQ4 (Security Code Generation):** Is the code generated by *ChatGPT* secure?

• **RQ5 (Non-determinism of *ChatGPT*):** How does the non-deterministic output of *ChatGPT* affect code generation?

Our experimental results demonstrate that (1) *ChatGPT* is better at generating functionally correct code for problems before 2021 in different languages than problems after 2021 with 48.14% advantage in *Accepted* rate on judgment platform, but *ChatGPT*'s ability to directly fix erroneous code with multi-round fixing process to achieve correct functionality is relatively weak; (2) the distribution of cyclomatic and cognitive complexity levels for code snippets in different languages varies. Furthermore, the multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets; (3) in algorithm scenarios with languages of C, C++, and Jave, and CWE scenarios with languages of C and Python3, the code generated by *ChatGPT* has relevant vulnerabilities. However, the multi-round fixing process for vulnerable code snippets demonstrates promising results, with more than 89% of vulnerabilities successfully addressed; and (4) code generation may be affected by *ChatGPT*'s non-determinism factor, resulting in variations of code snippets in functional correctness, complexity, and security.

Contributions. In summary, we make the following contributions to this paper:

• In this paper, we conduct a comprehensive empirical assessment to the quality of *ChatGPT*-based code generation;

• We systematically evaluate the *ChatGPT*-based code generation, including multi-round process, from three aspects: correctness, complexity, and security. The evaluated results reveal potential issues and limitations in *ChatGPT*-based code generation over the three aspects; and

• Our research contributes to advancing the potential knowledge and understanding of the capabilities of LLMs in enhancing software engineering practices, with a particular focus on code generation.

Online Artifact. The experimental scripts, results, and raw data are available at: [28].

2 Background

In this section, we briefly introduce LLMs, *ChatGPT*, and the use of *ChatGPT*.

LLMs and *ChatGPT*. LLMs (large language models) [6], [29], [30], [31], [32], [33], [20], [12] refer to a class of AI models that use an enormous amount of parameters and are designed to process and generate human-like text based on large-scale language datasets. These models utilize deep learning techniques, typically employing Transformer architectures [6], consisting of stacked encoders and decoders, to learn patterns, relationships, and structures in languages. Transformer utilizes self-attention mechanism to weigh the importance of words in the input text, capturing long-range dependencies and relationships between words. LLMs are trained on massive amounts of text data from various sources and show a strong ability in many NLP tasks, such as machine translation, question answering, summarization, and so on. *GPT* [32] and *BERT* [29] are based on the decoder (unidirectional) and encoder (bidirectional) components of the Transformer, respectively. They utilize pre-training and fine-tuning techniques. *GPT-2* [33] and *GPT-3* [20] are the successors of *GPT*, with *GPT-2* having a larger model size in parameters than *GPT*, and *GPT-3* being even larger than *GPT-2* with using 175 billion parameters. Additionally, with larger corpus, *GPT-2* and *GPT-3* introduce zero-shot and few-shot learning to enable adaptation to multitask scenarios. Moreover, *GPT-3* has demonstrated performance comparable to state-of-the-art fine-tuned systems across various tasks. *Codex* [9] is obtained by training *GPT-3* on GitHub code data. It serves as the underlying model for GitHub *Copilot* [11], a tool that can automatically generate

1. The version used in *ChatGPT* is *GPT-3.5* instead of *GPT-4*.

and complete code automatically. To enhance the alignment between LLMs and users (humans), *InstructGPT* [31] incorporates additional supervised learning and reinforcement learning from human feedback (RLHF) to fine-tune *GPT-3*. *ChatGPT* [12], [31], implemented atop *GPT-3.5* [21] (or *GPT-4* [22]), is now the most ideal product LLM that adapts to human expression by using Instruct [31]. *ChatGPT* utilizes the same methods as *InstructGPT* and provides the ability to answer follow-up questions (i.e., dialog ability) through RLHF. The dialog ability [12] enables *ChatGPT* to communicate with users conversationally, continuously generating information or correcting previously incorrect ones. This property makes *ChatGPT* even more powerful and versatile than previous LLMs. Thus, in this study, we take the state-of-the-art *ChatGPT* (the default version of *GPT-3.5*), the recent popular product, as the representative of LLMs for evaluation.

Use of *ChatGPT*. To use *ChatGPT*, developers send a text message as input. The message is called prompt, used to guide *ChatGPT*'s text generation. The prompt serves as a cue for the model to understand the desired output or the user's intent. *ChatGPT* responds based on the input prompt and the knowledge it learns from its massive amounts of training data. *ChatGPT* also supports answering follow-up questions (i.e., dialog ability), which allows users to engage in back-and-forth conversations. This capability enables users to send multiple text messages consecutively to *ChatGPT* and receive responses that maintain context and continuity.

When a user submits a series of messages to *ChatGPT*, each message within the conversation context is considered by the model when generating a response. The messages can include user prompts, system instructions, and previous responses from *ChatGPT* itself. By incorporating the conversation history, *ChatGPT* can produce responses that are more coherent and informed, taking into account the ongoing dialogue.

ChatGPT can be used in many software engineering tasks, including code generation. Additionally, by leveraging *ChatGPT*'s dialog ability, users can provide a series of prompts (i.e., multi-round process) for code generation (e.g., fix functionally incorrect code).

Example. For example, the prompt, *"Write a bubble sort program in Python to rank 1-10"*, can be used for *ChatGPT* to generate a bubble sort program in Python. As shown in Fig. 1, *ChatGPT* offers a sample code for the bubble sort program in Python, which fulfills the requirement given in the prompt. We can also append a new prompt *"Output the sorted result in nonincreasing order"* in the same conversation that *ChatGPT* is guided to replace comparison operator $>$ with $<$ in line 7 of the code snippet.

```
1  def bubble_sort(arr):
2      n = len(arr)
3      for i in range(n):
4          # Last i elements are already sorted
5          for j in range(0, n-i-1):
6              # Swap if the element found is greater
                    ↪ than the next element
7              if arr[j] > arr[j+1]:
8                  arr[j], arr[j+1] = arr[j+1], arr[j]
9  # The input list containing numbers from 1 to 10 in
        ↪ random order
10 numbers = [3, 7, 2, 8, 10, 1, 5, 9, 4, 6]
11 # Sort the numbers using bubble sort
12 bubble_sort(numbers)
13 # Print the sorted list
14 print("Sorted list of numbers from 1 to 10:")
15 print(numbers)
```

Fig. 1: *ChatGPT*-generated Bubble Sort Algorithm in Python.

3 Empirical Study Setup

In this section, we introduce the workflow of interacting with *ChatGPT* to generate code and other study setups.

3.1 Data Collection

The assessment of our study involves the utilization of two datasets: *LeetCode* problems [27] and CWE (Common Weakness Enumeration) scenarios (CWE's code scenarios) as provided in [23]. For *LeetCode* problem dataset, we randomly collect 728 algorithm problems where 354 and 374 of them are published after 2021 and before 2021, respectively. The reason for splitting them is because *ChatGPT* is trained on text data before 2021. For each problem, the problem description, input-output examples, and the method signature template in the specified language are used for code generation. As for the CWE scenario dataset, it contains 18 CWEs with 54 scenarios in *MITRE Top 25 CWEs* [34] (3 of them drop in rank to below 25 in 2022 *MITRE Top 25 CWEs*). For each CWE, three different code scenarios (context) are provided for code generation. The detailed introduction and preprocessing of these datasets is presented in the corresponding subsections in Sec. 4 (i.e., Sec. 4.1 and Sec. 4.4).

3.2 Methodology

Workflow. The overall workflow of our study framework is shown in Fig. 2. ❶ We construct a suitable prompt for the given *LeetCode* problem or CWE scenario (i.e., one CWE's code scenario) and send the constructed prompt to *ChatGPT*. ❷ *ChatGPT* generates a response based on the current round provided prompt and the previous round conversation context (first round has no previous round conversation context). We extract the code snippet by *ChatGPT* between two triple backticks from the response. ❸ For the generated code, we leverage *LeetCode* online judgment to test its functional correctness, or we utilize *CodeQL* [35] (with manual analysis) to detect CWE vulnerabilities. Here, we refer to them collectively as testing in Fig. 2. If the testing result passes (e.g., pass all test cases or no vulnerability detected), the code generation process ends. ❹ Otherwise, there are bugs (e.g., compile error) in the generated code snippet. If the (round) number in the conversation (i.e., dialog) with *ChatGPT* does not exceed the round limit (e.g., the maximum round number of 5), we utilize the feedback provided from *LeetCode* and *CodeQL* to reconstruct a new prompt and input it to *ChatGPT* for a new round of code generation (i.e., go back to ❶ for fixing). If the testing is consistently unpassed and the round number in the conversation exceeds the round limit, the code generation is considered failed. The entire process

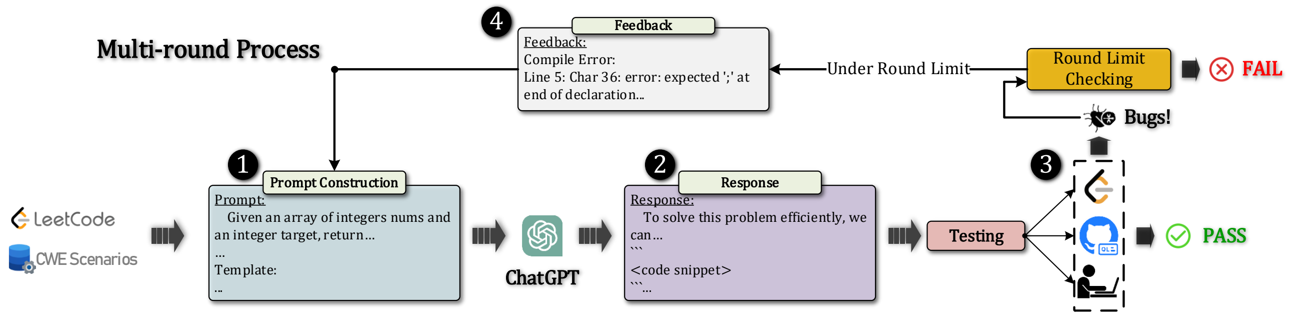


Fig. 2: The workflow of interacting with *ChatGPT* to generate code snippets.

including multiple rounds in the conversation is called the multi-round (fixing) process (one-round process with the maximum round number of 1 has no fixing property). The details of prompt construction, testing, and multi-round fixing process are explained in the subsections of Sec. 4.

Principle of Prompt Design. The goal of our prompt design is not to find the optimal prompt that maximizes *ChatGPT*'s performance. Instead, our goal is to provide a reasonable prompt that simulates real-world usage scenarios, especially for code generation, which can also avoid overfitting to the specific prompts and datasets. In developing the prompt template, we refer to online prompt templates (e.g., OpenAI Cookbook [36] and PromptBase [37]) for code generation tasks and finally establish the following principle for prompt design: *offering sufficient information to ChatGPT while leveraging its dialog ability*.

Subject LLM. The default language model provided by OpenAI [38] for *ChatGPT* is *GPT-3.5*. This model contains 175 billion parameters, making it a highly capable and complex model. *GPT-3.5* is engineered to handle a diverse range of natural language processing tasks, such as text generation, text completion, and other related tasks. In this study, we utilize the model version *gpt-3.5-turbo-0301* of *ChatGPT* for performing evaluation. We query *ChatGPT* through a simple wrapper [39] of OpenAI API [38] to easily control the dialog ability of *ChatGPT*. The temperature of *ChatGPT* is set to the default value of 0.7 [12] to simulate real-world usage scenarios. Furthermore, the token limitation of *ChatGPT* [12] is 4,096, which may influence the output from *ChatGPT*. If the total length of the input prompt and the generated response exceeds this limitation, then the excess part is discarded and possibly produces incomplete code snippets, causing errors in the generated code. In our experiments, we impose strict length limitations on both the input prompt and the generated response. For each round in the multi-round process, we find that the current round prompt[2] lengths and response lengths are all under 2,400 tokens and 800 tokens, respectively, which does not exceed *ChatGPT*'s token limitation. Thus, for the one-round process (e.g., Sec. 4.1), the outputs of *ChatGPT* are not influenced by the token limitation problem. However, in the complete multi-round process, especially when performing code generation for *LeetCode* problems, there are some cases where the token lengths used (include previous prompts, responses, and the current round prompt and response) can exceed the token limitation. To mitigate this issue when encountering the cases, we take a *token-limitation* strategy of adding necessary information (e.g., *LeetCode* problem descriptions) at the beginning of the current round prompt and remove as little of the beginning dialog content (in block granularity, i.e., one prompt or response) from the conversation as possible to keep the remaining token space for the response from *ChatGPT* having at least 1000[3] in length. This strategy avoids missing the necessary details in tasks for *ChatGPT*. Moreover, in our observation, the strategy guarantees that the generated code snippets are complete and at least ensures that the immediate previous round's response remains throughout the conversation such that *ChatGPT* does not lose the most recent code generation-related information. The detailed introduction of this strategy is presented in Sec. 4.2[4].

2. Disregard the prompts and responses of previous rounds.

3.3 Experiment Environment

All experiments are conducted on a server with an Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz (10 cores) and 128GB RAM. Its operating system is Ubuntu 20.04. The framework designed and scripts used in the experiments are developed in Python 3.10.9. *CodeQL* [35] used is in version 2.12.2.

4 Experiment and Evaluation

4.1 Functionally Correct Code Generation

RQ1: Is the code generated by *ChatGPT* functionally correct?

Motivation. Given an appropriate prompt, *ChatGPT* [12] is able to generate text consistent with the prompt based on knowledge learned. This ability may improve developer productivity [24], [40], [41], [42]. In the first step, we focus on evaluating the ability of *ChatGPT* to generate functionally correct code automatically in one-round process.

Approach. We let *ChatGPT* read the natural language description of the given problem to generate the corresponding code snippet in one-round process (i.e., the maximum round number is set to 1), and utilize the problems on *LeetCode* [27] as our dataset. *LeetCode* is an online platform that

3. In the first experiment in Sec. 4.1, all lengths of responses are under 770. Thus, We slightly amplify 770 to 1000 as the length of remaining token space that should be guaranteed when generating responses.

4. The tasks in Sec. 4.1 and Sec. 4.4 have no this issue. All token lengths used are lower than the token limitation.

provides challenging coding problems and automatic judgment. At the time of writing, there are over 2,500 problems on *LeetCode* with easy, medium, and hard levels, starting from the 2014 year. We collect all problems on *LeetCode*, and divide them into two categories, problems before 2021 (Bef. problems) and problems after 2021 (Aft. problems), by using the time divider of 2022-01-01. Since *ChatGPT* [12] is trained on text data before 2021, Bef. problems and corresponding solutions may have a high probability to appear in its training set. This case may degenerate the code generation task for Bef. problems into querying code in the database (i.e., code reuse [43], [4], [44]). Code reuse is a commonly used software development practice that avoids creating new code from scratch (e.g., copy-paste). Therefore, we take both problems into account.

Specifically, we focus on Algorithm problems[5] on *LeetCode* since Algorithm problems are the most significant, numerous, and diverse problems on the platform. The total numbers of Bef. problems and Aft. problems are 1,624 and 354, respectively. Furthermore, the difficulty level distribution to both of them is in the ratio of $1 : 2 : 1$ for hard, medium, and easy problems. Among all the Bef. problems, we sample 374 of them randomly, having similar quantities to the Aft. problems and following the same difficulty level distribution as Aft. problems. The ratio of the numbers of hard, medium, and easy problems is also $1 : 2 : 1$ for both 354 Aft. problems and 374 Bef. problems, consistent with the difficulty level distribution of all problems on the *LeetCode* platform. Additionally, we also check if there are significant differences between Bef. problems and Aft. problems. If Aft. problems are just reformulations of Bef. problems, *ChatGPT* may likely be able to easily solve them, which can affect the reliability of the experiment results in distinguishing between time periods. Specifically, we first use the "similar questions" provided for each problem on the *LeetCode* platform to find similar problem pairs of Bef. problems and Aft. problems. The "similar questions" [27] represent two paired problems that have similar scenarios (e.g., processing string) or require using similar algorithms for solving (e.g., dynamic programming). In total, there are 142 pairs found. Then, we have two graduate students independently and manually check these problem pairs. Through a careful checking and discussion process, we find that these similar problems are either having similar scenarios but completely different solution goals, or different scenarios and conditions but can be solved using similar algorithms such as dynamic programming. After a careful manual analysis, we do not find any cases that Bef. problems can be easily reformulated to obtain Aft. problems. Thus, we consider Aft. problems and Bef. problems to be sufficiently different. Moreover, for each problem, we ask *ChatGPT* to generate code in five different languages: C, C++, Java, Python3, and JavaScript. Moreover, we create a corresponding prompt using the same prompt template for each <*problem, language*> pair. In total, there are 1,870 and 1,770 prompts for Bef. problems and Aft. problems, respectively. Due to the rate-limiting of queries to *ChatGPT*, we input every prompt once into it to ask for generating code. Then, we submit parsed solutions to *LeetCode* for functional correctness judgment and get submission statuses [27] including *Accepted*, *Wrong Answer*, *Compile Error*, *Time Limit Exceeded*, and *Runtime Error*. They correspond to A., W.A., C.E., T.L.E., and R.E., respectively. One problem corresponds to one unique conversation to avoid triggering *ChatGPT*'s reasoning from other problems. The status explanations are as follows:

- *Accepted*: The submitted code snippet passes all test cases.
- *Wrong Answer*: The submitted code snippet has no compile errors but cannot pass all test cases.
- *Compile Error*: The submitted code snippet cannot be compiled.
- *Time Limit Exceeded*: The runtime of the submitted code snippet exceeds the permitted execution time.
- *Runtime Error*: The execution of the submitted code snippet triggers a runtime error for at least one test case.

Note that code with W.A. does not necessarily mean it does not contain R.E. or T.L.E.. Several errors can occur at the same time. Nevertheless, we study the functional correctness of code generation. Therefore, we take their priorities as C.E., R.E. $>$ W.A. $>$ T.L.E. by default, meaning that we only focus on the judgment results returned by *LeetCode* and this processing method has a negligible impact on the experimental conclusions. We prioritize C.E. and R.E. because these two errors lead to code running failures, which also implies W.A.. T.L.E. is set to the lowest priority because it mainly relates to non-functional requirements. Moreover, *LeetCode* online judgment platform terminates the testing process upon encountering the first failed test case. Thus, the test case pass rates (the percentage of predefined test cases that a submitted code snippet successfully passes) provided by the platform may serve as a lower bound.

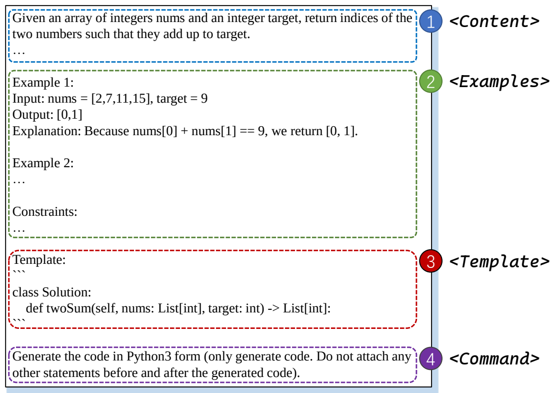
We evaluate *ChatGPT*'s ability of code generation on the metric of status rate (SR) defined as follows:

$$SR = \frac{N_c}{N_i} \times 100\% \qquad (1)$$

Where, N_c and N_i are the number of code snippets generated belonging to the status and the number of prompts input, respectively. Status is either A., W.A., C.E., T.L.E., or R.E.. The deep analysis for code with W.A., C.E., T.L.E., or R.E. is presented in Sec. 4.2.

We also conduct Wilcoxon rank-sum test [45] and Cliff's Delta effect size measure [46] to compare two independent samples and determine whether there are significant differences between them and quantify the magnitude of the differences observed between the two independent samples. The null hypothesis for the Wilcoxon rank-sum test is that there is no significant difference between the two samples where the samples are the combinations of SR values in different conditions (e.g., A. rate values of five languages in different period problems). If the obtained p-value from Wilcoxon rank-sum test is small (less than 0.05), it suggests that there is a statistically significant difference between the two independent samples. In cases of multiple comparisons, we apply Holm-Bonferroni correction [47], a commonly used technique, to adjust p-values to reduce the risk of Type I errors. The absolute value of effect size (effect size value) obtained from Cliff's Delta measure ranges from 0 to

5. https://leetcode.com/problemset/algorithms/



Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.
…

Example 1:
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:
…

Constraints:
…

Template:
```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
```

Generate the code in Python3 form (only generate code. Do not attach any other statements before and after the generated code).

Fig. 3: An example of prompt for two sum problem in Python3.

```c
int maximumGroups(int* grades, int gradesSize){
    ...
    qsort(grades, gradesSize, sizeof(int), cmpfunc);
        ↪ // sort grades in ascending order
    ...
}

int cmpfunc (const void * a, const void * b) {...}
```

Fig. 4: Function in C code generated by *ChatGPT* is not declared before invocation.

1. A value close to 0 indicates a small effect, meaning that there is minimal difference between the two independent samples, and a value close to 1 indicates a substantial effect size, meaning that there are significant differences between them. By combining the results from the Wilcoxon rank-sum test and Cliff's Delta, we can gain a comprehensive insight into the differences in code generation results, allowing us to draw more robust conclusions.

Prompt. The prompt template designed consists of 4 components. They are <*Content*>, <*Examples*>, <*Template*>, and <*Command*>, aligning with the principle of prompt design (see Sec. 3.2). Fig 3 shows an example of a prompt. <*Content*> describes the problem in nature language, <*Examples*> shows <*input, output*> pairs of functionally correct code, <*Template*> specifies the method signature of generated code, and <*Command*> asks for generating code in a specific language.

Result. Table 1 and 2 show code generation results judged by *LeetCode* for five languages in two periods and in two forms, SR, and corresponding relative frequency bar chart. Columns of Python3 and JavaScript contain no C.E. since both of them are dynamic programming languages. From the overall results, *ChatGPT* generates functionally correct code for Bef. problems at a significantly higher A. rate than Aft. problems. Specifically, the average A. rate (68.41%) in five languages of Bef. problems exceeds Aft. problems' (20.27%) by 48.14%. The performance in five languages of code generation in different periods is significantly different with a p-value of 0.008 and an effect size value of 1.

• **Aft. Problems.** For Aft. problems, the overall A. rate is lower than 25%, where the A. rates of hard, medium, and easy problems are 0.66%, 13.90%, and 52.47%, respectively. The p-values adjusted using Holm-Bonferroni correction

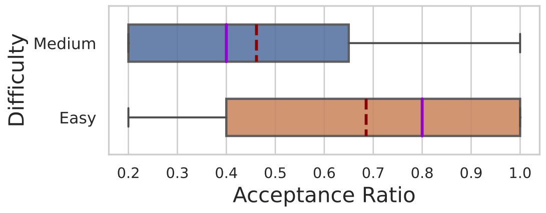


Fig. 5: Distribution of the ratios of languages accepted to corresponding Aft. problems. Where dark violet and dark red lines represent the median and mean, respectively.

procedure and effect size values between different difficulties in five languages are all less than 0.05 and equal to 1, respectively. The result indicates that *ChatGPT*'s ability to functionally correct code generation decreases significantly as the difficulty of the problem increases in the face of Aft. problems. Additionally, even for easy problems, it is only able to answer half of them correctly. Out of these five/four metrics, the W.A. rate is the highest one reaching 58% for all languages. Moreover, each W.A. code snippet has an average of 109 test cases, however, the code generated by *ChatGPT* can pass only 25% of them. Hard, medium, and easy problems achieve 20.90%, 21.03%, and 38.41% test case pass rates, respectively. Thus, regardless of the difficulty, the semantics of the code generated differs significantly from the logic of the corresponding problem descriptions. In addition, the C.E. rate and R.E. rate also reach 16%, and hard and medium problems' rates are significantly higher than easy problems. The code generated by *ChatGPT* for hard and medium problems is more likely to contain both compile and runtime errors. The compile errors include undeclared variable, function declaration error, uninitialized variable, constant function (i.e., generate an empty body), and so on. For example, Fig. 4 shows that the generated function `cmpfunc` is not declared before invocation. The syntax errors account for only a small fraction (3.7%) of these errors. For runtime errors, there are null pointer dereference, out-of-bound, heap-buffer-overflow, type error, and so on, which are common in human-written code. As for T.L.E. rate, it does not dominate a high value (6%), but the average pass rate of test cases is 51% which is higher than W.A. code snippets'. The average test case pass rates of three difficulty levels in hard, medium, and easy of T.L.E. problems are 68%, 50%, and 1% (easy problems can be neglected due to their T.L.E. rate close to 0%), respectively. Since T.L.E. code snippets' test case pass rate is partial, it is the lower bound for these problems, and at most, an additional 6% of the generated code can be functionally correct, even though their time complexity may not be ideal.

Breaking down to each language, language C, C++, Java, Python3, and JavaScript have A. rates of 15.38%, 19.37%, 20.17%, 23.93%, and 22.51%, respectively. Moreover, the A. rate distributions (acceptance ratio distributions) of combining five different languages to each problem (only consider problems have at least one correct solution) are shown in Fig. 5. From the figure, we can see that both medium's mean and median lines are $\leq$ 0.5, and easy's ones are all $\geq$ 0.6. *ChatGPT* is easier to generalize code generated to different languages for easy problems. The differences between easy and medium's median and mean are 0.4 and

TABLE 1: Code-judged Result in C, C++, and Java Languages

Period	Difficulty	C					C++					Java				
		A.	W.A.	C.E.	T.L.E.	R.E.	A.	W.A.	C.E.	T.L.E.	R.E.	A.	W.A.	C.E.	T.L.E.	R.E.
Aft.	Hard	0.00 %	62.64 %	27.47 %	2.20 %	7.69 %	0.00 %	62.64 %	26.37 %	7.69 %	3.30 %	0.00 %	71.11 %	17.78 %	5.56 %	5.56 %
	Medium	5.88 %	45.29 %	28.82 %	7.06 %	12.94 %	13.37 %	57.56 %	13.37 %	6.98 %	8.72 %	15.12 %	69.19 %	7.56 %	5.81 %	2.33 %
	Easy	48.89 %	36.67 %	10.00 %	0.00 %	4.44 %	51.14 %	45.45 %	1.14 %	1.14 %	1.14 %	50.00 %	41.11 %	4.44 %	2.22 %	2.22 %
	Total	15.38 %	47.58 %	23.65 %	3.99 %	9.40 %	19.37 %	55.84 %	13.68 %	5.70 %	5.41 %	20.17 %	62.50 %	9.38 %	4.83 %	3.13 %
Bef.	Hard	18.28 %	38.71 %	33.33 %	3.23 %	6.45 %	42.55 %	19.15 %	31.91 %	2.13 %	4.26 %	52.81 %	26.97 %	13.48 %	2.25 %	4.49 %
	Medium	46.51 %	20.35 %	20.93 %	2.91 %	9.30 %	70.62 %	14.12 %	11.86 %	0.00 %	3.39 %	78.16 %	12.07 %	4.60 %	3.45 %	1.72 %
	Easy	76.29 %	9.28 %	6.19 %	0.00 %	8.25 %	89.22 %	6.86 %	1.96 %	0.98 %	0.98 %	94.06 %	3.96 %	1.98 %	0.00 %	0.00 %
	Total	47.24 %	22.10 %	20.17 %	2.21 %	8.29 %	68.63 %	13.40 %	14.21 %	0.80 %	2.95 %	76.37 %	13.46 %	6.04 %	2.20 %	1.92 %
-	Total	31.28 %	34.64 %	21.88 %	3.09 %	8.84 %	44.75 %	33.98 %	13.95 %	3.18 %	4.14 %	48.74 %	37.57 %	7.68 %	3.49 %	2.51 %

TABLE 2: Code-judged Result in Python3 and JavaScript Languages

Period	Difficulty	Python3				JavaScript			
		A.	W.A.	T.L.E.	R.E.	A.	W.A.	T.L.E.	R.E.
Aft.	Hard	2.20 %	80.22 %	12.09 %	5.49 %	1.10 %	83.52 %	5.49 %	9.89 %
	Medium	18.13 %	61.40 %	11.11 %	9.36 %	16.96 %	67.84 %	9.94 %	5.26 %
	Easy	57.30 %	34.83 %	2.25 %	5.62 %	55.06 %	38.20 %	1.12 %	5.62 %
	Total	23.93 %	59.54 %	9.12 %	7.41 %	22.51 %	64.39 %	6.55 %	6.55 %
Bef.	Hard	45.56 %	33.33 %	8.89 %	12.22 %	42.22 %	46.67 %	2.22 %	8.89 %
	Medium	79.07 %	16.28 %	1.16 %	3.49 %	80.47 %	13.02 %	1.78 %	4.73 %
	Easy	95.96 %	4.04 %	0.00 %	0.00 %	93.07 %	5.94 %	0.00 %	0.99 %
	Total	75.35 %	17.17 %	2.77 %	4.71 %	74.44 %	19.44 %	1.39 %	4.72 %
-	Total	50.00 %	38.06 %	5.90 %	6.04 %	48.80 %	41.63 %	3.94 %	5.63 %

```c
int** matrixReshape(int** mat, int matSize, int*
    ↪ matColSize, int r, int c, int* returnSize,
    ↪ int** returnColumnSizes){
    // calculate the total number of elements...
    int totalElements = matSize * matColSize[0];
    // check the number of elements...
    if (r * c != totalElements) {...}
    // allocate memory for the new matrix
    int **newMat = (int **)malloc(sizeof(int *) * r);
    for (int i = 0; i < r; i++) {
        newMat[i] = (int *)malloc(sizeof(int) * c);
    }
    // copy the elements from the original matrix...
    ...
    // set the return values
    *returnSize = r;
    *returnColumnSizes = (int *)malloc(sizeof(int));
    (*returnColumnSizes)[0] = c;
    return newMat;
}
```

Fig. 6: Code generated by *ChatGPT* triggers heap-buffer-overflow runtime error.

0.22, respectively. Moreover, the average acceptance rate in humans for the problems accepted by *ChatGPT* is 66%, and the one for the problems unaccepted by *ChatGPT* is 48%. *ChatGPT* has similarities with human performance. In addition, functionally correct code's runtime and memory overheads are excellent in the human ranking, averaging over 68% and 51% solutions, respectively.

- **Bef. Problems.** As for Bef. problems, the A. rates of hard, medium, and easy problems are 40.13%, 70.95%, and 89.80%, respectively, which are much higher than Aft. problems' though there still exist significant differences among different difficulties. The p-values adjusted using Holm-Bonferroni correction procedure and effect size values between hard and medium and hard and easy difficulties

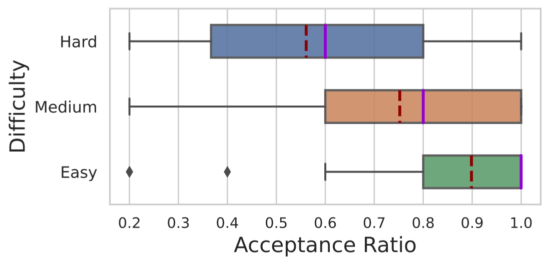


Fig. 7: Distribution of the ratios of languages accepted to corresponding Bef. problems (the meaning of two lines is presented in Fig. 5).

in five languages are all less than 0.05 and greater than 0.9, respectively. The adjusted p-value and effect size value between medium and easy difficulties in five languages are 0.056 and 0.76, respectively. *ChatGPT* performs better against problems that may appear in the training set before 2021, especially for medium and easy problems. The A. rate of solving hard problems has increased by 40% but is still below 50%, which indicates *ChatGPT*'s ability to generate code for logically complex problems still has a big room for improvement. The overall W.A. rate decreases to 17.03%, and hard, medium, and easy problems' W.A. rates are 32.89%, 15.05%, and 6%, respectively. The code generated can still pass 25% of average 112 test cases. Hard, medium, and easy problems achieve 19.19%, 31.12%, and 47.32% test case pass rates, respectively. Both the latter two have a 10% improvement, which indicates that *ChatGPT* has a better understanding of Bef. problems. However, the C.E. rate and R.E. rate still reach 13% close to Aft. problems' 16% with a p-value and effect size value, between two periods, of 0.328 and 0.3125, respectively, and hard problems have the highest rate, followed by the medium ones. The compile errors and runtime errors are similar to Aft. problems' including undeclared variable, uninitialized variable, null pointer dereference, out-of-bounds, heap-buffer-overflow, type error, and so on. For example, the code shown in Fig. 6 is used to reshape a given 2-dimensional matrix but triggers runtime errors at line 15 that allocates a wrong size of memory to `*returnColumnSizes`. To T.L.E. rate, the value decreases to 1.87% with an average of 74% test case pass rate.

Breaking down to each language, C, C++, Java, Python3, and JavaScript have A. rates of 47.24%, 68.63%, 76.37%, 75.35%, and 74.44%, respectively. The rate values of the last four languages are close to each other and substantially higher than the rate value of C, the lowest-level language, for at least 20%. Fig. 7 shows the same as Fig. 5 but for Bef. problems. From the figure, we can see that medium and easy's mean and median lines are ≥ 0.75, and the differences between their median and mean are smaller than previous ones of Aft. problems by half. Moreover, hard's mean and median lines are both ≥ 0.55. *ChatGPT* is easier to generalize code to different languages for Bef. problems. The average acceptance rate in humans for the problems accepted by *ChatGPT* is 55%, and the one for the problems unaccepted by *ChatGPT* is 47%. Functionally correct code's runtime and memory overheads are also excellent in the human ranking, averaging over 71% and 54% solutions, respectively.

We also sample 50 problems from all problems (25 Aft. problems and 25 Bef. problems, and each problem has 5 solutions in 5 different languages) to investigate how many times *ChatGPT* accurately generates the exact solutions (token-by-token) to ground truth solutions. We collect 5 distinct ground truth solutions to each *ChatGPT*-generated solution and the ground truth solutions are obtained from the LeetCode platform and [48][6]. By our manual analysis, we find that none of the solutions are generated on a token-by-token basis. However, for Bef. problems we find that 6 solutions in easy and medium difficulties are Type-2 Clone (i.e., have some renamed unique identifiers) [49] to ground truth solutions. This result indicates that *ChatGPT* may have a certain probability (>5%) of reproducing similar solutions from the training set for Bef. problems, especially in easy and medium difficulties.

Answer to RQ1: Functionally Correct Code Generation

❶ *ChatGPT* is better at generating functionally correct code for Bef. problems in different languages than Aft. problems. Specifically, the average A. rate of the former exceeds the one of the latter by 48.14%. Additionally, different levels of difficulty also have an impact on *ChatGPT*-based code generation;
❷ For both problems, *ChatGPT* is able to generate code with smaller runtime and memory overheads than at least 50% human solutions;
❸ Regardless of the periods of problems, *ChatGPT* has a similar probability of 14.23% on average to generate code with compile or runtime errors; and
❹ The A. rate values of C++, Java, Python3, and JavaScript with 44.75%, 48.74%, 50.00%, and 48.80% are close to each other and substantially higher than the rate value of C with 31.28% for all problems.

6. It is notable that the analysis result is a lower bound since it is impossible to check all the ground truth for each solution.

TABLE 3: The Statistics of 157 <*problem, language*> Pairs

	Hard	Medium	Easy	Total
C	5	12	5	22
C++	5	17	7	29
Java	6	22	5	33
Python3	8	16	8	32
JavaScript	9	22	10	41
Total	33	89	35	157
Problem	12	25	13	50

4.2 Multi-round Fixing for Code Generation

RQ2: How effective is the multi-round fixing process in improving code generation for functional correctness?
Motivation. *ChatGPT* supports multiple rounds of conversations (i.e., dialog ability), and users can use this feature to continuously generate code to get functionally correct code in one conversation. In this RQ, we study the multi-round fixing process for fixing code snippets with W.A., C.E., R.E., or T.L.E. errors.
Approach. We ask *ChatGPT* in multiple rounds of conversation to fix the error code snippets. Moreover, before analyzing the results of the multi-round fixing process, we also further manually analyze the causes of each category of errors including W.A., C.E., T.L.E., and R.E. for the code generated by *ChatGPT* for a better understanding. The exact procedures for analyzing different categories of errors are shown in the ***Analyzing*** part in each following subsection.
Result. We first analyze the corresponding errors and then apply multi-round fixing for code generation (multi-round code generation).

4.2.1 Code with Wrong Answer

► ***Analyzing:*** We randomly choose 50 problems with W.A. code snippets to analyze the causes of *Wrong Answer*. In total, there are 157 <*problem, language*> pairs and among these 50 problems, there is an average of 3.14 pairs per problem that are in W.A.. The basic statics of these pairs are shown in Table 3. To analyze the causes, two graduate students with experience in algorithm analysis manually check each W.A. code snippet and assign each of them to one defect class. During this process, if there is any disagreement, the two students further discuss it with a senior software analyst to resolve the disagreement. Out of these 157 pairs, 115 pairs receive consistent classifications and the consistency ratio is 0.7325. The remaining 42 pairs reach a unanimous classification through discussion.

We utilize the defect categories from the fixing perspective used in [50] (CodeFlaws) and [51]. [50] analyzes the programs submitted in Codeforces [52] and classifies the defects in these programs into multiple classes, and [51] follows the defect classification in [50] to construct the code defects generated by *Codex* [9]. The defect classes, definitions, and classified results are shown in Table 4. As we can see, most of the defects fall under the M-L subclass of Multi-hunk and Misaligned Algorithm subclass of Algorithm-related with 51 and 62, respectively. The other subclasses of defects are relatively much less, especially for S-A, S-F, S-AS, and S-DS which are all 1. From the perspective of difficulty, hard problems require more fixing work than medium and easy

TABLE 4: Defect Classification of the 157 <*problem, language*> Pairs

Defect Class	Subclass	Definition	Logic	Hard	Medium	Easy	Total
Multi-hunk	Similar (M-S)	Similar single-hunk defects are at multiple discontinuous locations of code snippets.	WD, MCC	0	2	4	6
	Unique (M-U)	Diverse single-hunk defects are at multiple discontinuous locations of code snippets, and the total lines of fixes are no more than five lines.	WD, MCC	0	3	6	9
	Need Large Fix (M-L)	The defect is neither M-S nor M-U and needs to edit more than five lines at multiple locations of code snippets.	MCC	9	32	10	51
	Code Block (M-B)	The fix of defect needs moving or exchanging code blocks with few (<5) additional statements inserted, updated, or deleted.	WD, MCC	0	2	0	2
Single-hunk	Operator Mutation (S-O)	Replace arithmetic/logical/relational/bitwise operator with another operator or insert/delete operators and relevant operands or modify operator precedence.	WD	0	2	6	8
	Array Mutation (S-A)	Replace the array access with other constant/variable, operands with arithmetic operators, or replace an array with another array.	WD	0	1	0	1
	Function Call Mutation (S-F)	Replace function call with another function call or change function arguments.	WD	0	0	1	1
	Add Statements (S-AS)	Insert a continuous chunk of statements.	WD	0	0	1	1
	Delete Statements (S-DS)	Delete a continuous chunk of statements.	WD	0	1	0	1
	Higher Order (S-HO)	A single-hunk patch that combines multiple single-hunk bugs.	WD	0	13	2	15
Algorithm-related	Misaligned Algorithm	The algorithm used is misaligned with the requirement given in the problem description.	MP	24	33	5	62
Total	-	-	-	33	89	35	157

```
1   class Solution:
2       def smallestNumber(self, pattern:  str) ->  str:
3           n =  len(pattern)
4           result = [0] * (n + 1)
5           curr_min, curr_max = 0, n
6           for i in  range(n):
7               if pattern[i] == 'I':
8                   result[i] = curr_min
9                   curr_min += 1
10              else:
11                  result[i] = curr_max
12                  curr_max -= 1
13          result[n] = curr_min
14          ans = ''
15          for i in  range(n + 1):
16              ans +=  str(result[i] + 1)
17          return ans
```

Fig. 8: An example code snippet with WD error stemming from a misunderstanding of the meaning of lexicographically smallest in the given problem description.

```
char *categorizeBox(...) {
    ...
    if (...) {
        category = "Bulky";
    }
    else if (...) {
        category = "Heavy";
    }
    else {
        category = "Neither";
    }
    if (strcmp(category, "Bulky") == 0 &&
      ↪ strcmp(category, "Heavy") == 0) {
        category = "Both";
    }
    else if (strcmp(category, "Bulky") == 0 &&
      ↪ strcmp(category, "Heavy") != 0) {
        category = "Bulky";
    }
    else if (strcmp(category, "Heavy") == 0 &&
      ↪ strcmp(category, "Bulky") != 0) {
        category = "Heavy";
    }
    return category;
}
```

Fig. 9: An example code snippet with WD error stemming from the generated code that is not consistent with the understanding of the problem.

ones since hard problems have only M-L and Misaligned Algorithm while medium and easy ones have other subclasses. Moreover, during our manual analysis of these code snippets, we also find that the causes of errors from logic perspective could be divided into three categories as follows (also see Table 4):

① **Wrong Detail (WD)**: The code generated by *ChatGPT* has errors in some details. These detail errors stem from a little misunderstanding (e.g., a word) of the given problem or the generated code that is not consistent with the understanding of the problem. Fig. 8 and 9 show two code examples corresponding to the two detail errors, respectively. Fig. 8 is an example of having a little misunderstanding of the given problem. The given problem 2375[7] is a medium one asks to generate the lexicographically smallest possible string that meets conditions. However, the generated code does not completely hold the meaning of lexicographically smallest due to line 11 decreasing number from the maximum one. As for the code snippet in Fig. 9, it is an example of the generated code that is not consistent with the understanding of the problem. The given problem 2525[8] is an easy one that asks to categorize a given box into

7. https://leetcode.com/problems/construct-smallest-number-from-di-string/

8. https://leetcode.com/problems/categorize-box-according-to-criteria/

```
1 int* findArray(...){...
2     arr[0] = pref[0];
3     for (int i = 1; i < prefSize; i++) {
4         arr[i] = arr[i-1] ^ pref[i];
5     }
6     *returnSize = prefSize;
7     return arr;
8 }
```

Fig. 10: An example code snippet with MCC error that reasons a wrong recurrence formula.

```
public class Solution {
    public int totalSteps(int[] nums) {
        int steps = 0;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i - 1] > nums[i]) {
                steps++;
                nums[i] = nums[i - 1];}}
        return steps;}}
```

Fig. 11: An example code snippet with MP error.

one of the categories according to its properties. *ChatGPT* understands the meaning of the problem but fails to transfer the meaning of the problem description in natural language to code semantics. This is reflected in conditional expressions of lines 12, 15, and 18. We take line 12 as an example. The condition in natural language is "If the box is both "`Bulky`" and "`Heavy`", then its category is `Both`", and *ChatGPT* utilizes `strcmp` twice to compare if the `category` string is both "`Bulky`" and "`Heavy`". The meaning of the natural language description is not equivalent to the code semantics.

WD errors are also easy to be fixed by humans since the generated code logic is roughly correct. The defect subclasses corresponding to WD errors are mainly the subclasses other than M-L and Misaligned Algorithm, based on our manual analysis.

② **Misunderstanding Certain Content (MCC)**: The code generated by *ChatGPT* does not hold the main condition of the given problem. However, the algorithm used by the generated code is suitable. Fig. 10 shows an example code snippet. The corresponding problem 2433[9] is a medium one and asks to find the solution satisfying one `xor`-based condition. The key to this problem is to solve the correct recurrence formula according to this `xor`-based condition. *ChatGPT* reasons out a recursive formula in lines 2-5, but this recursive formula does not satisfy the condition required by the problem. Other typical examples are the problems using dynamic programming (DP) that the generated code uses wrong DP equations.

MCC errors are more difficult to fix than WD errors by humans since the core of the code needs to be modified to meet conditions provided by problems. The defect class corresponding to MCC errors is Multi-hunk based on our manual analysis.

③ **Misunderstanding Problem (MP)**: *ChatGPT* misunderstands or does not understand the problem description given. The generated code does not hold all conditions and uses wrong (misaligned) algorithms. Fig. 11 shows an example for the problem 2289[10].

MP errors are the most difficult to fix among these three kinds of errors by humans since the code needs rewriting completely. The defect subclass corresponding to MP errors is Misaligned Algorithm based on our manual analysis.

► ***Multi-round Fixing:*** We take each code snippet of <*problem, language*> pair to *ChatGPT* to continuously generate code in one unique conversation with multiple rounds. The round limit number is set to 5, providing a reasonable maximum number of fixes to 5 times [53]. For each pair, we create an initial prompt by leveraging the corresponding problem (i.e., <*Content*> and <*Examples*> in Fig. 3), the code snippet, and error message. The error message is returned by *LeetCode* online judgment, which is suitable to be taken as feedback provided to *ChatGPT*. One example is shown below (where bolded words are filled in according to each pair's information):

Prompt:
Given four integers length, width, height, and...

The code in **C** below cannot pass all test cases:
\`\`\`
char *categorizeBox(...) {...}
\`\`\`

Error Message:
Last test case: 2909 3968 3272 727
Code output: "Bulky"
Expected output: "Both"

Fix the code and generate the fixed code.

If the newly generated code snippet is still not accepted (i.e., W.A, C.E., R.E., and T.L.E.), the corresponding error message is taken directly as a new prompt provided to *ChatGPT* to fix and generate a new code snippet, in the same conversation. It is appropriate to use the error message directly as the new prompt since *ChatGPT* has the ability to dialog. The whole process lasts for a maximum of five rounds (one round corresponds to one newly generated code snippet) if the generated code is never accepted. However, there are cases that the cumulative token length of previous prompts, responses, and the current round prompt and response exceeds the token limitation of *ChatGPT*. We mitigate this problem with *token-limitation* strategy (see Sec. 3.2) through reusing the initial prompt template with the current round's error code and message, which avoids missing necessary information of problem description to *ChatGPT*. Moreover, it guarantees completely generated code snippets and also at least remains the immediate previous round's response in practice (Sec. 3.2).

The result of multi-round code generation is shown in Table 5, where '/'s left hand and right hand represent the accepted (i.e., code snippets accepted in five rounds) number and the total number, respectively. From the result, we can see that the majority of these 157 <*problem, language*>

9. https://leetcode.com/problems/find-the-original-array-of-prefix-xor/description/

10. https://leetcode.com/problems/steps-to-make-array-non-decreasing/description/

TABLE 5: Result of Multi-round Code Generation for W.A. Code Snippets

	Hard	Medium	Easy	Total
C	0/5 (0.0 %)	0/12 (0.0 %)	1/5 (20.0 %)	1/22 (4.5 %)
C++	0/5 (0.0 %)	1/17 (5.9 %)	3/7 (42.9 %)	4/29 (13.8 %)
Java	0/6 (0.0 %)	0/22 (0.0 %)	3/5 (60.0 %)	3/33 (9.1 %)
Python3	0/8 (0.0 %)	3/16 (18.8 %)	4/8 (50.0 %)	7/32 (21.9 %)
JavaScript	2/9 (22.2 %)	3/22 (13.6 %)	5/10 (50.0 %)	10/41 (24.4 %)
Total	2/33 (6.1 %)	7/89 (7.9 %)	16/35 (45.7 %)	25/157 (15.9 %)
Problem	2/12 (16.7 %)	6/25 (24.0 %)	12/13 (92.3 %)	20/50 (40.0 %)

TABLE 6: Result of 10-round Code Generation for 10 W.A. Code Snippets

Problem Title	Language	Difficulty	Result
Make Array Zero by Subtracting Equal Amounts	C++	Easy	W.A.
Maximum Enemy Forts That Can Be Captured	JavaScript	Easy	W.A.
Construct Smallest Number From DI String	JavaScript	Medium	A.
Construct Smallest Number From DI String	Python3	Medium	R.E.
Frog Jump II	JavaScript	Medium	W.A.
Longest Nice Subarray	Java	Medium	W.A.
Minimum Number of Steps to Make Two Strings Anagram II	JavaScript	Medium	A.
Make the XOR of All Segments Equal to Zero	Java	Hard	R.E.
Maximum Number of Points From Grid Queries	JavaScript	Hard	W.A.
Minimum Difference in Sums After Removal of Elements	C	Hard	W.A.

pairs cannot be fixed by automation. Only 25 pairs are fixed in 5 different languages, and 16 of them are problems at easy level. The pairs at medium level are fixed with only 7 pairs though its total number of pairs is more than twice as many as the pairs at easy level. Pairs at hard level are nearly impossible to be fixed. The percentage of fixes for pairs under all difficulties is less than half. However, judging from the fixes of the problems, 12 out of 13 easy problems are fixed. The percentage of fixes for hard and medium problems is still below 30%. The average number of rounds per fixed pair is 1.32. 21 of the 25 can be fixed with just one round. The defect classes of the 25 pairs are mostly Multi-hunk, where M-S, M-U, M-L, and M-B account for 3, 3, 12, and 1, respectively. The redundant is in Single-hunk and Algorithm-relate, where S-O, S-AS, S-HO, and Misaligned Algorithm account for 2, 1, 2, and 1, respectively.

To further analyze why most of the code snippets cannot be fixed under the multi-round process, we randomly select 10 more pairs from these unfixed pairs and expand the round limit number to 10 for multi-round fixing. The results are shown in Table 6. In these 10 code snippets, 2 of them are successfully fixed under 10 rounds. The remaining 8 still fail to be fixed, including 2 that are eventually fixed as R.E.. We manually check these failed pairs' final generated code snippets and find that 7 of them marked as dark grey deviate significantly from the meaning of the corresponding problems (i.e., MCC and MP). The remaining one marked as light grey is almost correct, with only a very small logical error (i.e., WD and single-hunk), where this error persists throughout the multi-round process. Moreover, there are only 5 W.A. code snippets with single-hunk fixed under the previous 5-round fixing.

Therefore, we conclude that there are 2 reasons why *ChatGPT* cannot automatically fix W.A. code snippets through multi-round fixing. On one hand, *ChatGPT* lacks the ability to grasp logical details, even though these details may be straightforward for humans. *ChatGPT* struggles to notice and make corresponding fixes to them. Thus, *ChatGPT* needs improving for its implementation ability for logical details. On the other hand, *ChatGPT* lacks in dealing with problems that require complex reasoning (for W.A. code snippets), resulting in the code newly generated still deviating from the actual meaning of the problems. As a result, these kinds of W.A. code snippets are difficult to be fixed directly and automatically, but it is not always the case (e.g., <*Construct Smallest Number From DI String, Python3*> and <*Construct Smallest Number From DI String, JavaScript*> in Table 6 where the latter one is fixed at the 10-th round).

★ **Summary 1.** Most of the defects of code with W.A. fall under the M-L subclass and Misaligned Algorithm subclass with 51 and 62, respectively. The other subclasses of defects are relatively much less.

★ **Summary 2.** After our manual analysis, we conclude that W.A. code snippets can be divided into three categories of WD, MCC, and MP, from logic perspective.

★ **Summary 3.** By applying multi-round fixing, *ChatGPT* has difficulty fixing W.A. code snippets. We conclude for two reasons: (1) *ChatGPT* lacks the ability to grasp logical details, and (2) *ChatGPT* lacks in dealing with problems that require complex reasoning.

4.2.2 Code with Compile Error

► ***Analyzing:*** We analyze all C.E. code snippets and classify them manually based on the compile error messages returned by *LeetCode*. There are 312 code snippets with C.E. in three different languages, C, C++, and Java.

The compile error classes, explanations, and classified results are shown in Table 7. From the table, we can see that the majority of compile errors are in the class of constant function, accounting for half (159/314) of all compile errors. The code snippets having constant function compile error means that the functions or methods in code snippets have empty body (i.e., the generated codes are the same as corresponding code templates provided). Thus, this type of compile error is not a real compile error for generated code since it is a case of failure of code generation by *ChatGPT* (it is not a failure of response). For other three special compile errors of classes of wrong method name, redefinition of `main`, and incompatible parameter types, they are related to *LeetCode* online judgment platform, inconsistent with the settings in *LeetCode* but not real compile errors. For example, a compile error-free code snippet generated by *ChatGPT* may contain *main* function but *LeetCode* has set another internal *main* for running test cases, which causes compile error of redefinition of `main`. Nevertheless, for wrong method name and incompatible parameter types, though they do not indicate real compile errors, it shows that *ChatGPT* may have a certain chance to generate code regardless of the requirement (i.e., code template) given in the prompt. More interestingly, we also find that for code snippets with compile error of wrong method name, a few method names used for Aft. problems are method names of Bef. problems. For example, problem 2449[11] requires using `makeSimilar` as method name but *ChatGPT* generates method name of `minOperations` which is used in problem 1658[12], which may point to inference attack problem [54], [55]. As for the remaining classes of compile errors, they are

11. https://leetcode.com/problems/minimum-number-of-operations-to-make-arrays-similar/
12. https://leetcode.com/problems/minimum-operations-to-reduce-x-to-zero/

TABLE 7: Compile Error Classification of All C.E. Code Snippets

Class	Explanation	Hard	Medium	Easy	Total
Label error	A label can only be part of a statement and a declaration is not a statement.	0	0	1	1
Redefinition	A symbol has been defined in multiple places.	0	4	1	5
Function declaration error	Function is invoked before declaration.	4	19	1	24
Undeclared variable	Variable is referenced or used in a program without being previously declared or defined.	7	5	2	14
Wrong method name	Use a different method (or function) name than the template provided by *LeetCode* to define a method (or function).	8	2	2	12
Constant function	Function (or method) is generated with empty body.	81	71	7	159
Redefinition of `main`	Generated code snippet contains `main` function already provided by *LeetCode*.	10	12	7	29
Use undeclared function	Generated code snippet uses undeclared and undefined functions (mainly caused by EBL).	18	25	3	46
Incompatible parameter types	Mismatch between the expected parameter type and the actual argument type passed to a method or function.	2	0	1	3
Uninitialized variable	Variable is being declared and initialized but not given a valid initial value before using.	2	2	0	4
Invalid operators	Invalid operands to binary operators (e.g., `%`, `+`, `/`, and so on).	2	1	0	3
Error of `#include`	Indicated file cannot be compiled.	1	4	0	5
Syntax error	Generated code snippet has syntax errors.	2	4	0	6
No attribute	The accessed member does not exist in the corresponding data structure.	1	2	0	3
Total	-	138	151	25	314

the real compile errors not triggered by *LeetCode* platform. Table 7 provides the explanations of corresponding classes of these compile errors, and examples of these classes can be found at our online artifact [28].

► ***Multi-round Fixing:*** We follow the settings in W.A.'s multi-round fixing. The prompt used in C.E.'s multi-round fixing has a little bit different from the previous one. One example is shown below:

```
Prompt:
Write a function to find the longest...

The code in C below has compile errors:
```
char *longestCommonPrefix(...) {...}
```

Error Message:
Compile Error
solution.c: In function 'longestCommonPrefix'
Line 22: Char 5: error: a label can only be part of...
char *prefix = (char *)malloc((prefixLen + 1) * sizeof(char));
^ ~~~

Fix the code and generate the fixed code.
```

Where the error message also comes from *LeetCode* online judgment. The entire fixing process continues until the generated code snippet is accepted or the process reaches the maximum round number of 5. We take the final status (e.g., A.) in one conversation as the final generation result for the corresponding <*problem, language*> pair. The strategy of mitigating token limitation follows the setting in W.A. multi-round fixing. In addition, we do fixing for all classes except for the class of constant function, since fixing constant function is equivalent to regenerating the entire code snippets for <*problem, language*> pairs.

TABLE 8: Result of Multi-round Code Generation for C.E. Code Snippets

Class	Hard	Medium	Easy
Label error	-	-	0:1:0
Redefinition	-	3:0:1	1:0:0
Function declaration error	3:1:0	14:4:1	1:0:0
Undeclared variable	6:0:1	3:0:2	2:0:0
Wrong method name	8:0:0	2:0:0	2:0:0
Redefinition of `main`	9:0:1	8:0:4	6:0:1
Use undeclared function	12:3:3	16:6:3	3:0:0
Incompatible parameter types	2:0:0	-	1:0:0
Uninitialized variable	2:0:0	2:0:0	-
Invalid operators	1:0:1	1:0:0	-
Error of `#include`	0:1:0	3:1:0	-
Syntax error	1:1:1	1:1:2	-
No attribute	1:0:0	1:0:1	-
Total	45:6:7	54:12:14	16:1:1

The result is shown in Table 8. The *x*:*y*:*z* in the table is the generation result under different conditions, where *x*, *y*, and *z* represent the number of fixed code snippets (i.e., C.E. → A., W.A., or T.L.E.), the number of errors retained in code snippets (i.e., C.E. → the same C.E.), and

```
1 ...
2 char* longestCommonPrefix(...) {
3     ...
4     goto exit_loop;
5     ...
6 exit_loop:
7     char* prefix = ...;
8     ...}
```

Fig. 12: An example code snippet in C with retained error (label error) of EIL. The code snippet is the final generated one in the conversation.

```
int* smallestTrimmedNumbers(...){
    ...
    qsort(nums, numsSize, sizeof(char*), cmp);
    ...}
```

Fig. 13: An example code snippet in C with changed error (use undeclared function) of EIL. The code snippet is the final generated one in the conversation.

the number of errors changed in code snippets (i.e., C.E. → other C.E. or R.E.), respectively. From the result, we can see that most of the code snippets can be fixed. 19 and 22 code snippets in C and C++ get retained errors and changed errors, respectively. For the 115 fixed code snippets, 40 of them are accepted, containing 30, 7, and 3 in C, C++, and Java, respectively. For the 40 code snippets, their classes of compile errors contain redefinition (1), function declaration error (8), undeclared variable (1), wrong method name (2), redefinition of `main` (8), use undeclared function (12), incompatible parameter types (1), uninitialized variable (3), invalid operators (1), and error of `#include` (1). As for the code snippets with retained errors and changed errors, we manually analyze them and divide the causes of the two errors into two categories as follows:

① **Errors in Languages (EIL)**: EIL errors arise from the properties of the language used (i.e., C and C++) to implement the code. The errors include both retained errors and changed errors. Fig. 12 shows an example code snippet of retained error (label error), where the code snippet is the final generated one in the conversation. In this particular case, the retained error is related to label error, where the code violates the rule of C by placing a label (line 6) before a declaration (line 7). *ChatGPT* fails to fix the error in 5 rounds even though the error message contains the explanation of label error shown in Table 7. Regarding changed errors, all of them are runtime error, except one no attribute error and two use undeclared function errors, which means that almost all C.E. errors are fixed. *ChatGPT* tries to generate functionally correct code, but R.E. errors are triggered in the implementation of algorithms. For instance, the final generated code may have an out-of-bound error. We further discuss R.E. subsequently. For the two classes of C.E. errors, we show an example of use undeclared function in Fig. 13 that *ChatGPT* fixes its original syntax error on '}' but introduces another C.E. error. The specific error is related to the comparison function `cmp` used as an argument for the `qsort` function. However, `cmp` is not defined in the code snippet, resulting in a compile error.

② **Errors between Languages (EBL)**: Different from EIL,

```
#include <cstring>
#include <algorithm>
...
char *subStrHash(...) {...}
```

Fig. 14: An example code snippet in C with retained error (error of `#include`) of EBL. The code snippet is the final generated one in the conversation.

EBL errors arise from the similarity between languages of C and C++. The errors still include both retained errors and changed errors, and all changed errors are C.E. errors which are the same as the C.E. errors in retained errors of EBL. Thus, we only use the retained error as an example. Fig. 14 shows an example code snippet of retained error (error of `#include`). The language for this <*problem, language*> pair is C but the generated code uses `<cstring>` and `<algorithm>` header files belonging to C++.

We observe the entire multi-round process for these unfixed <*problem, language*> pairs of which the final errors are C.E. errors. We find that in most cases, although the error messages provided contain the causes of the C.E. errors and the corresponding locations, and *ChatGPT* is aware of the error problem from its natural language, the newly generated code snippets still have the same errors. One example is Fig. 12 that *ChatGPT* notices the error in each round but fails to fix it. For these errors, a potentially appropriate approach is to add information to prompts from human knowledge that triggers *ChatGPT* to truly fix errors. For instance, for the example of Fig. 14, we can supply extra information (e.g. "the code snippet is in language C, you cannot use C++ header files") to *ChatGPT* to fix the error.

Additionally, for each <*class, difficulty*> (e.g., <*use undeclared function, medium*>) in Table 8 except for label error, it has at least one code snippet can be fixed. Thus, *ChatGPT*'s multi-round code fixing for errors (include R.E. errors. See Table 10) may also be related to the randomness (i.e., Temperature) of *ChatGPT* itself or the code snippets it receives.

★ **Summary 1.** More than half of *ChatGPT*'s C.E. errors (in static languages) are unreal compile errors, especially for constant function, wrong method name, and incompatible parameter types. This experimental result indicates that *ChatGPT*'s code generation stability (avoid generating empty body) and alignment with human attention (meet user requirements such as method signature provided) are the potentially severe issues that need to be strengthened.

★ **Summary 2.** By applying multi-round fixing, most (70%) of C.E. code snippets can be fixed, including 26% of them can be fixed to A.. After analyzing unfixed code snippets, it can be inferred that the unfixed reasons can be concluded to EIL and EBL. Additionally, a potential approach to help *ChatGPT* fix the unfixed errors is to add human knowledge.

4.2.3 Code with Runtime Error

► ***Analyzing:*** We also analyze all R.E. code snippets and classify them manually based on the runtime error messages returned by *LeetCode*. There are 194 code snippets with R.E. in five different languages, C, C++, Java, Python3, and JavaScript.

The runtime error classes, explanations, and classified results are shown in Table 9. From the table, we can see that

TABLE 9: Runtime Error Classification of All R.E. Code Snippets

Class	Explanation	Hard	Medium	Easy	Total
Integer-overflow	The result of an arithmetic operation exceeds the maximum value that can be represented by a given integer data type (e.g., `int`).	4	17	2	23
Heap-buffer-overflow	A program writes data beyond the allocated memory block in the heap, potentially overwriting adjacent data or causing a crash due to corrupted memory.	9	22	8	39
Undefined-behavior	The executing of a certain code statement is not defined or unpredictable, potentially leading to unexpected runtime errors or crashes (e.g., left shift of negative value).	1	1	1	3
Out-of-bound	Access an array, list, or other data structure by using an index or pointer that exceeds the valid range of elements or memory.	18	21	4	43
Constant function	Function (or method) is generated with empty body.	3	6	1	10
Null pointer dereference	A program attempts to access or manipulate data through a null pointer, which is a pointer that does not point to a valid memory location.	4	8	2	14
Wrong method name	Use a different method (or function) name than the template provided by *LeetCode* to define a method (or function).	3	4	6	13
Type error	An operation is performed on an object of an incompatible type (e.g., try to concatenate a string with an integer in Python3).	9	10	2	21
Value error	A method (function) or operation receives a valid type of input, but the value itself is not suitable or within the expected range for the specific operation (e.g., pass an empty *List* to function `max` in Python3).	0	4	1	5
Heap-use-after-free	Use a memory block in the heap after it has been deallocated (freed).	1	1	0	2
Recursion error	A function or method calls itself recursively without reaching a base case or termination condition.	4	3	0	7
Uninitialized variable	Variable is being declared and initialized but not given a valid initial value before using.	1	1	0	2
Syntax error	Generated code snippet has syntax errors.	3	0	0	3
Undeclared variable	Variable is referenced or used in a program without being previously declared or defined.	2	2	0	4
No attribute	The accessed member does not exist in the corresponding data structure.	0	3	0	3
Divided by zero	Attempt to divide a number by zero.	0	2	0	2
Total	-	62	105	27	194

```
1   class Solution:
2       def largestInteger(self, num:  int) -> int:
3           num_str = str(num)
4           ...
5           while swapped:
6               ...
7               for i in range(n):
8                   for j in range(i+1, n):
9                       if (num_str[i] % 2 == num_str[j]
                            ↪ % 2) and (num_str[i] <
                            ↪ num_str[j]):
10                          ...
11                  ...
12          return int(num_str)
```

Fig. 15: An example code snippet in Python3 with type error.

there are a small number of code snippets in the class of constant function for dynamic languages (i.e., Python3 and JavaScript), which is different from C.E.'s results. Moreover, like C.E. errors, wrong method name in R.E. errors is not a real runtime error and we also find examples that Aft. problems use method names of Bef. problems (e.g., problem 2164[13] and problem 905[14]). Additionally, the majority of runtime errors are overflow runtime errors (i.e., integer-overflow, heap-buffer-overflow, and out-of-bound) and the languages used in these errors are also mainly in static languages, C, C++, and Java, which is similar to humans making runtime errors. For dynamic languages (i.e., Python3 and JavaScript), the majority of runtime errors are in the class of type error. The error occurs when an operation is performed on an object of an incompatible type. One example is shown in Fig. 15 that in line 9, the code statement performs modulus operations (`%`) on characters `num_str[i]`

13. https://leetcode.com/problems/sort-even-and-odd-indices-independently/
14. https://leetcode.com/problems/sort-array-by-parity/

TABLE 10: Result of Multi-round Code Generation for R.E. Code Snippets

Class	Hard	Medium	Easy
Integer-overflow	2:2:0	16:1:0	2:0:0
Heap-buffer-overflow	5:1:3	14:8:0	5:3:0
Undefined-behavior	0:1:0	1:0:0	1:0:0
Out-of-bound	14:3:1	15:6:0	4:0:0
Null pointer dereference	3:1:0	6:2:0	1:1:0
Wrong method name	1:0:2	4:0:0	6:0:0
Type error	8:1:0	8:1:1	2:0:0
Value error	-	4:0:0	0:1:0
Heap-use-after-free	0:0:1	1:0:0	-
Recursion error	4:0:0	3:0:0	-
Uninitialized variable	1:0:0	1:0:0	-
Syntax error	3:0:0	-	-
Undeclared variable	2:0:0	2:0:0	-
No attribute	-	2:0:1	-
Divided by zero	-	2:0:0	-
Total	43:9:7	79:18:2	21:5:0

```
1  class Solution:
2      def minOperationsToFlip(...) -> int:
3          ...
4          def evaluate(mapping):
5              stack = []
6              for char in expression:
7                  if char == ')':
8                      ...
9                  elif char in {'0', '1', '&', '|',
                       ↪ '(', ')'}:
10                     stack.append(mapping[char])
11             ...
12         ...
13         # count the number of unique variables
14         variables = set([char for char in expression
               ↪ if char.isalpha()])
15         # generate all possible mappings ...
16         mappings = []
17         for i in range(2**n):
18             mapping = {}
19             for j, var in enumerate(variables):
20                 mapping[var] = (i >> (n-j-1)) & 1
21             mappings.append(mapping)
22         ...  # later invoke evaluate(mapping)
```

Fig. 16: An example code snippet in Python3 with a retained error (value error).

and num_str[j], which is not valid. As for the remaining classes, their explanations are provided in Table 9 and their examples can be found at our online artifact [28].

▶ ***Multi-round Fixing:*** We follow the settings in C.E.'s multi-round fixing, and the prompt template used here is modified, turning "The code in <*language*> below has compile errors:" to "The code in <*language*> below has runtime errors:".

The result is shown in Table 10. The x:y:z in the table is the same as Table 8. From the result, we can see that most of the code snippets can be fixed, and there are 32 and 9 code snippets in five different languages that get retained errors and changed errors, respectively. For the 143 fixed code snippets, 52 of them are accepted, containing 23, 12, 2, 10, and 5 in C, C++, Java, Python3, and JavaScript, respectively. For the 52 code snippets, their classes of runtime errors contain integer-overflow (11), heap-buffer-overflow (12), undefined-behavior (1), out-of-bound (10), null pointer dereference (4), wrong method name (3), type error (5), value error (1), heap-use-after-free (1), recursion error (1), uninitialized variable (2), and divided by zero (1). As for the code snippets with retained errors and changed errors, they are few in number (41) and most of them belong to overflow error. Regarding retained errors, by manual analysis, we believe that the main reason why the runtime errors cannot be eliminated is the algorithm implementation by *ChatGPT*. It is similar to those (e.g., WD) in the W.A. errors. One example of overflow is shown in Fig. 6 that *ChatGPT* fails to fix line 15 under the 5 rounds of dialogue. Fig. 16 shows an example of value error. The problem (or conflict) in the code snippet is between line 14 and line 9 (i.e., `char.isalpha()` and `char in '0', '1', '&', '|', '(', ')'`). The `char.isalpha()` condition checks whether a character is an alphabetic character (a-z or A-Z), while the `char in '0', '1', '&', '|', '(', ')'` condition checks for specific characters which are not alphabetic. Regarding the 9 changed errors, 1 is changed to compile error (use undeclared function) and the remaining errors are still runtime errors including heap-buffer-overflow (1), undefined-behavior (2), out-of-bound (2), type error (2), and recursion error (1). These new errors are introduced as the code snippets continue to be fixed and some parts of code snippets conflict.

We also observe the entire multi-round process for these unfixed <*problem, language*> pairs of which the final errors are R.E. errors. Like C.E., in most cases, *ChatGPT* notices errors based on the error messages provided, however, the newly generated code snippets still have the same errors. Fig. 6 and Fig. 16 are two examples. To fix runtime errors, also like C.E. errors, a potentially appropriate approach is to add information to prompts from human knowledge. For instance, for fixing Fig. 6, we can supply extra information (e.g., "`*returnColumnSizes = (int *)malloc(sizeof(int));` allocates a wrong size of memory to `*returnColumnSizes`") to fix the error.

★ **Summary 1.** The majority of runtime errors for static languages and dynamic languages are overflow (105) and type error (21), respectively. Additionally, for Python3 and JavaScript, 10 code snippets are constant function and 13 are wrong method name. The result indicates that *ChatGPT*'s code generation stability and alignment with human attention are also potentially severe issues for dynamic languages.

★ **Summary 2.** By applying multi-round fixing, like C.E., most (78%) of R.E. code snippets can be fixed, including 28% of them can be fixed to A.. After analyzing unfixed code snippets, it can be concluded that *ChatGPT* is flawed in the details of the algorithm implementation and a potential approach to trigger *ChatGPT* to fix the unfixed errors is to add human knowledge.

4.2.4 Code with Time Limit Exceeded

▶ ***Analyzing:*** There are 140 code snippets with T.L.E. in five different languages, C, C++, Java, Python3, and JavaScript. Two graduate students together analyze each T.L.E. code snippet and categorize all these code snippets based on the

```
bool reachingPoints(int sx, int sy, int tx, int ty){
    while (tx >= sx && ty >= sy) {
        ...
        if (tx > ty) {
            tx -= ty;
        } else {
            ty -= tx;
        }
    }
    return false;
}
```

Fig. 17: An example code snippet in C with AIAI.

```
class Solution:
    def countSpecialNumbers(self, n:  int) ->  int:
        def is_special(x):
            digits =  set()
            while x > 0:
                digit = x % 10
                ...
                x //= 10
            return True
        count = 0
        for i in  range(1, n+1):
            if is_special(i):
                count += 1
        return count
```

Fig. 18: An example code snippet in Python3 with CMA. The time complexity of the code snippet is $O(n \log n)$.

analysis results. After manually analyzing these code snippets, we classify their timeout reasons into three categories:

① **Aligned but Inefficient Algorithm Implementation (AIAI)**: The algorithm used in code generated by *ChatGPT* is aligned with the requirement given in the problem description, but some parts are not efficient. One classic example of AIAI is `gcd` functions, which use the modulo operator `%` or subtraction operator `-`. Both two functions with either the modulo operator or subtraction operator use the Euclidean algorithm, but their time complexities are $O(\log n)$ and $O(n)$, respectively. Some T.L.E. errors are caused by AIAI. Fig. 17 shows an example[15] similar to *gcd*. It uses subtraction operator `-` rather than modulo operator `%`, which increases the time complexity and fails to pass all test cases in the limited time set by *LeetCode* platform.

② **Functionally Correct but Misaligned Algorithm (CMA)**: The algorithm used by *ChatGPT* is functionally correct to the corresponding problem but it is inefficient for the limited time set by *LeetCode* platform. So, the algorithm is misaligned. In CMA, the most common examples of the generated code are to solve problems using the brute-force method. Although the algorithm (or code) using the brute-force method is functionally correct, the time complexity may also be very high (e.g., $O(2^n)$ time complexity), and thus the code is judged as timeout by *LeetCode* online judgment. One example[16] is shown in Fig. 18 whose time complexity is exponential (the input size of the code is $\log n$, representing the input number n).

15. https://leetcode.com/problems/reaching-points
16. https://leetcode.com/problems/count-special-integers/

```
1   var maximumGroups = function(grades) {
2       grades.sort((a, b) => b - a);
3       let groups = 0;
4       while (grades.length > 0) {
5           let sum = 0;
6           let count = 0;
7           for (let i = 0; i < grades.length; i++) {
8               if (sum + grades[i] <= count) {
9                   sum += grades[i];
10                  count++;
11                  grades.splice(i, 1);
12                  i--;
13              }
14          }
15          groups++;
16      }
17      return groups;
18  };
```

Fig. 19: An example code snippet in JavaScript with IA.

TABLE 11: Result of Multi-round Code Generation for T.L.E. Code Snippets

	Hard	Medium	Easy	Total
C	2/3 (66.7 %)	7/17 (41.2 %)	0/0 (0.0%)	9/20 (45.0 %)
C++	2/9 (22.2 %)	1/12 (8.3 %)	2/2 (100.0 %)	5/23 (21.7 %)
Java	2/8 (25.0 %)	8/16 (50.0 %)	0/2 (0.0 %)	10/26 (38.5 %)
Python3	3/19 (15.8 %)	7/21 (33.3 %)	0/2 (0.0 %)	10/42 (23.8 %)
JavaScript	2/8 (25.0 %)	8/20 (40.0 %)	0/1 (0.0 %)	10/29 (34.5 %)
Total	11/47 (23.4 %)	31/86 (36.0 %)	2/7 (28.6 %)	44/140 (31.4 %)

③ **Functionally Incorrect Algorithm (IA)**: The algorithm used by *ChatGPT* is functionally incorrect (i.e., WD, MCC, and MP) to the corresponding problem, which is also slow (due to the incorrect function) to resolve given problem instances (i.e., test cases). The algorithm may be aligned or misaligned (e.g., brute-force method). One example[17] is shown in Fig. 19, using greedy algorithm. The algorithm is aligned, but its function is incorrect. In line 8, the condition `sum + grades[i] <= count` can lead the outer while loop to be an infinite loop if the condition is `false`, causing T.L.E. error.

► ***Multi-round Fixing:*** We follow the settings in W.A.'s multi-round fixing.

The result is shown in Table 11. There are 44/140 code snippets across different languages and difficulty levels in total that can be fixed (i.e., accepted) by *ChatGPT*. By manual analysis, we find that these code snippets are in AIAI, CMA, and IA. For code snippets in AIAI and IA, *ChatGPT* tends to generate patches or rewrite code in different algorithms for fixing. One example of generating patches is Fig. 17 that *ChatGPT* modifies subtraction operator `-` to modulo operator `%`. The example of rewriting code is that for problem 1047[18] in C, *ChatGPT* changes aligned stack-based algorithm to array-based algorithm without using stack. As for code snippets in CMA, *ChatGPT* tends to change the algorithms used. As for the remaining 96 code snippets not fixed, their newly generated code snippets are judged as W.A. (64), R.E.

17. https://leetcode.com/problems/maximum-number-of-groups-entering-a-competition/
18. https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/

(8), and T.L.E. (24) by *LeetCode* online judgment. For the new ones judged as W.A., *ChatGPT* is able to fix the conflict parts causing T.L.E. but the incorrect functions cannot be fixed (e.g., Fig. 19), or it changes the algorithms used but the new algorithms are functionally incorrect (e.g., Fig. 18 in Java version). For the new ones judged as R.E., the runtime errors of them include integer-overflow, heap-buffer-overflow, out-of-bound, value error, and out-of-memory[19]. For the new ones judged as T.L.E., 16 of them can pass $> 75\%$ test cases and only 4 of them pass $< 50\%$ test cases. By our manual analysis, we find that *ChatGPT* tends to change algorithms used to avoid T.L.E., but the algorithms used have some inefficient parts (i.e., AIAI or IA). Thus, to fix these inefficient parts, a potentially appropriate approach is to tell *ChatGPT* the inefficient locations and fixing suggestions by humans.

★ **Summary 1.** By manually analyzing T.L.E. code snippets, it can be concluded that the timeout reasons are AIAI, CMA, and IA.

★ **Summary 2.** By applying multi-round fixing, only 31% code snippets can be fixed. For fixed code snippets in AIAI and IA, *ChatGPT* tends to generate patches or rewrite code in different algorithms, and for code snippets in CMA, *ChatGPT* tends to change the algorithm used. The unfixed ones have 24 in T.L.E., caused by AIAI and IA. To fix them, a potential approach is to provide inefficient locations and fixing suggestions by humans.

Answer to RQ2: Multi-round Fixing for Code Generation

❶ Multi-round fixing process can only fix a small fraction ($< 32\%$) of code snippets with W.A., C.E., R.E., or T.L.E. to A.. For fixing code snippets with C.E., R.E., or T.L.E. to A., W.A., or T.L.E. (exclude T.L.E. $\rightarrow$ T.L.E.), most ($\geq 70\%$) of them can be fixed by using multi-round fixing;
❷ Our analysis identifies several factors for errors in code snippets and unfixed cases under the multi-round fixing process. The findings contribute to the ongoing research focused on improving functionally correct code generation.

4.3 Code Complexity

RQ3: How complex is the code generated by *ChatGPT*?
Motivation. The complexity of code is a critical factor influencing code readability, maintainability, and overall quality [56], [24], [57]. In this RQ, we evaluate the complexity of the code generated by *ChatGPT*.
Approach. We utilize *SonarQube* [58] and *cccc* [59] to calculate two metrics for evaluating the complexity of Bef. and Aft. generated code, including the code generated in multi-round fixing. The metrics are cyclomatic complexity and cognitive complexity [24], [56], [60], and their specific meanings are as follows:

- **Cyclomatic Complexity:** The complexity counts the number of linearly independent paths through a given source code. It determines how difficult the given code is to test. A high cyclomatic complexity can potentially lead to a high probability of errors and bugs.
- **Cognitive Complexity:** The complexity refers to a measure of how difficult it is to understand and reason about a piece of code from the human perspective. It takes factors into account like control structures but slightly different from cyclomatic complexity. Its specific methodology can be found in [61]. A high cognitive complexity can affect code maintainability and increase the risk of bugs or errors.

19. A computer program tries to allocate memory from the heap, but there is not enough available memory to fulfill the request

TABLE 12: Cyclomatic Complexity Result of Generated Code

Language	Cyclomatic Complexity Level	Number of Code
C	Low complexity (1-4)	148 (21.9 %)
	Moderate complexity (5-7)	223 (33.0 %)
	High complexity (8-10)	140 (20.7 %)
	Very High complexity (11+)	165 (24.4 %)
C++	Low complexity (1-4)	218 (34.2 %)
	Moderate complexity (5-7)	241 (37.8 %)
	High complexity (8-10)	90 (14.1 %)
	Very High complexity (11+)	89 (13.9 %)
Java	Low complexity (1-4)	225 (33.7 %)
	Moderate complexity (5-7)	240 (36.0 %)
	High complexity (8-10)	100 (15.0 %)
	Very High complexity (11+)	102 (15.3 %)
Python3	Low complexity (1-4)	314 (44.5 %)
	Moderate complexity (5-7)	226 (32.1 %)
	High complexity (8-10)	105 (14.9 %)
	Very High complexity (11+)	60 (8.5 %)
JavaScript	Low complexity (1-4)	221 (32.5 %)
	Moderate complexity (5-7)	234 (34.4 %)
	High complexity (8-10)	115 (16.9 %)
	Very High complexity (11+)	111 (16.3 %)

TABLE 13: Cognitive Complexity Result of Generated Code

Language	Cognitive Complexity Level	Number of Code
Java	Low complexity (<5)	244 (36.6 %)
	Moderate complexity (6-10)	231 (34.6 %)
	High complexity (11-20)	147 (22.0 %)
	Very High complexity (21+)	45 (6.7 %)
Python3	Low complexity (<5)	301 (42.6 %)
	Moderate complexity (6-10)	211 (29.9 %)
	High complexity (11-20)	147 (20.8 %)
	Very High complexity (21+)	47 (6.7 %)
JavaScript	Low complexity (<5)	250 (36.7 %)
	Moderate complexity (6-10)	235 (34.5 %)
	High complexity (11-20)	146 (21.4 %)
	Very High complexity (21+)	51 (7.5 %)

Where cognitive complexity is measured for three languages (Java, Python3, and JavaScript) due to the limitation of *SonarQube* and *cccc*.

We also utilize *LeetCode* solutions [48] in C++ and Python3 written by humans (lack solutions in other languages) to compare with *ChatGPT*'s, for observing their different extent in code complexity.

Note that the complexity is measured in terms of problems as a unit (most solutions have only one method).
Result. We first examine code snippets not generated in multi-round fixing. The analysis for code snippets generated in multi-round fixing is discussed in **Multi-round Comparisons** in this section. Table 12 and 13 show the cyclomatic and cognitive complexity values of code generated by *ChatGPT* in five languages. Table 14 and 15 show the cyclomatic and cognitive complexity values of code written by humans in two languages.

• **Cyclomatic Complexity.** Based on the official documentation of PMD [62], cyclomatic complexity can be categorized into four classes low (1-4 cyclomatic complexity value), moderate (5-7), high (8-10), and very high complexity ($\geq$

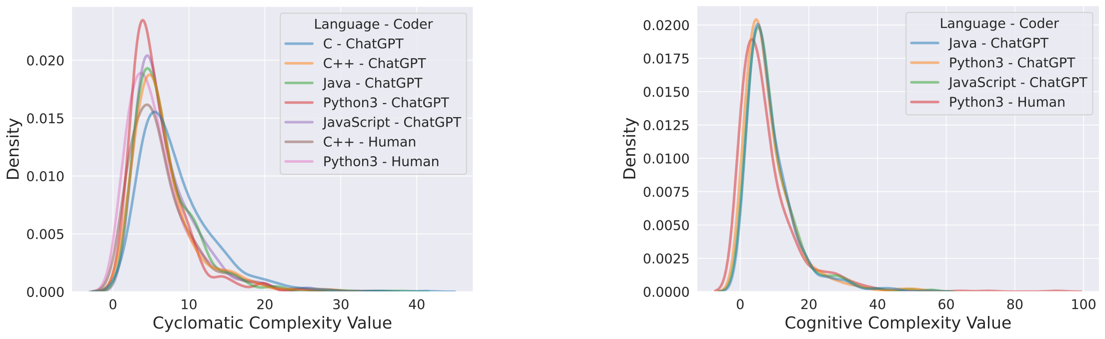


Fig. 20: Density graph of cyclomatic and cognitive complexity.

TABLE 14: Cyclomatic Complexity Result of Written Code

Language	Cyclomatic Complexity Level	Number of Code
C++	Low complexity (1-4)	243 (39.3 %)
	Moderate complexity (5-7)	203 (32.8 %)
	High complexity (8-10)	84 (13.6 %)
	Very High complexity (11+)	89 (14.4 %)
Python3	Low complexity (1-4)	323 (45.8 %)
	Moderate complexity (5-7)	205 (29.1 %)
	High complexity (8-10)	87 (12.3 %)
	Very High complexity (11+)	90 (12.8 %)

TABLE 15: Cognitive Complexity Result of Written Code

Language	Cognitive Complexity Level	Number of Code
Python3	Low complexity (<5)	357 (50.6 %)
	Moderate complexity (6-10)	176 (25.0 %)
	High complexity (11-20)	112 (15.9 %)
	Very High complexity (21+)	60 (8.5 %)

11). From Table 12, we can find that both low and moderate complexities dominate more than 50% in five languages for generated code, with C having the lowest percentage at 54.9%, while other languages exceed 66%. The difference is at least 11%. For the other four languages, their complexity distributions are similar. Specifically, the differences between the maximum and minimum of four complexity levels from low to high are 12%, 5.7%, 2.8%, and 7.8%. When excluding Python3, the differences decrease to 1.7%, 3.4%, 2.8%, and 2.4%. Notably, Python3 has a much higher percentage (44.5%) of low complexity and a much lower percentage (8.5%) of very high complexity, while C++, Java, and JavaScript have more similar distributions of the four complexity levels.

Compared with human solutions in C++ and Python3 (see Table 14), we find that the complexity distributions of the generated code for both languages closely resemble those of the written code. For C++, the written code's percentage of low complexity is 5% higher than the generated code's, and correspondingly, the one of moderate complexity is 5% lower than the generated code's. They have similar percentages of high and very high complexities. As for Python3, both generated code and written code have similar percentages of low complexity, while the percentages of moderate and high complexities of generated code are higher than written code's by 3% and 2.6%. Consequently, the former's percentage of very high complexity is lower than the latter's with 4.3%.

• **Cognitive Complexity.** According to [61], cognitive complexity can also be categorized into four classes low (<5 cognitive complexity value), moderate (6-10), high (11-20), and very high complexity ($\geq$ 21). From Table 13, we can see that both low and moderate complexities dominate more than 70% in five languages. The generated code in Java and JavaScript has similar distributions of four complexity levels. For Python3, it has a 6% higher percentage (42.6%) of low complexity, and correspondingly, its percentage of moderate complexity is lower than the other two languages' with 4.7%. Python3's percentages of high and very high complexities are also similar to those in Java and JavaScript.

Compared with human solutions in Python3 (see Table 14), we find that the written code has a higher percentage of low complexity with 8% than the generated code. Correspondingly, the former's percentages of moderate and high complexity are lower than the latter's with double 4.9%, respectively. However, the difference between the two percentages of very high complexity for the two kinds of code is only 2%.

★ **Summary 1.** Fig. 20 also further shows the density graphs of cyclomatic and cognitive complexities for each language and corresponding coder (i.e., *ChatGPT* and *Human*) pair. The horizontal coordinate is the complexity value and the vertical one is the corresponding density. By analyzing the two figures, we can gain a more intuitive insight that *ChatGPT*'s Python3's distributions have comparatively smaller mean, while C's ones have larger mean. The distributions of C++, Java, and JavaScript are nearly overlapping. In addition, the distributions of code written by humans skew to the left compared with code generated in corresponding languages by *ChatGPT*. Therefore, we can conclude that the level of complexity of code generated by *ChatGPT* varies among the five programming languages. The generated code in C is more complex than the other languages, while the complexity of code in C++, Java, and JavaScript is comparable. The code in Python3 is the least complex. Moreover, the complexity of code generated by *ChatGPT* in C++ and Python3 is slightly higher but nearly equal to that of code written by humans.

• **Cross-difficulty Comparisons.** We further examine the distributions of cyclomatic complexity and cognitive complexity under different difficulty levels. The results are shown in Fig. 21 and Fig. 22. Each row in the figures corresponds to the percentages of the same complexity level

TABLE 16: Comparision between Code Generated by *ChatGPT* and Code Written by Humans to the Distributions of Cyclomatic and Cognitive Complexities under Three Difficulty Levels of Problems

Metric	Level	Easy Difficulty				Medium Difficulty				Hard Difficulty			
		C++	C++-H	Python	Py-H	C++	C++-H	Python	Py-H	C++	C++-H	Python	Py-H
Cyclomatic	Low	46.2%	56.5%	67.6%	69.0%	34.3%	39.0%	43.4%	46.0%	17.3%	16.3%	22.0%	21.2%
	Moderate	37.6%	31.5%	25.0%	21.9%	38.5%	34.3%	34.8%	33.0%	36.7%	31.1%	34.5%	29.1%
	HIgh	10.2%	9.2%	4.3%	8.0%	13.8%	15.3%	16.2%	11.2%	20.1%	15.6%	23.7%	19.0%
	Very High	5.9%	2.7%	3.2%	1.1%	13.5%	11.3%	5.6%	9.7%	25.9%	37.0%	19.8%	30.7%
Cognitive	Low	-	-	63.8%	73.3%	-	-	40.4%	51.3%	-	-	24.2%	25.7%
	Moderate	-	-	22.9%	18.2%	-	-	34.2%	27.4%	-	-	29.2%	27.4%
	HIgh	-	-	11.7%	8.0%	-	-	21.5%	16.5%	-	-	29.2%	22.9%
	Very High	-	-	1.6%	0.5%	-	-	3.8%	4.7%	-	-	17.4%	24.0%

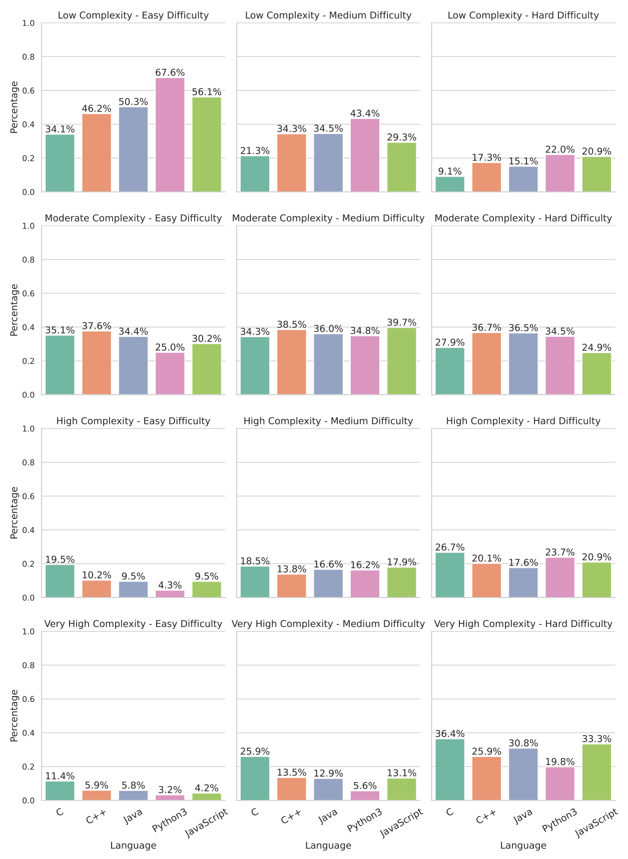


Fig. 21: Distribution of cyclomatic complexity under three difficulty levels of problems.

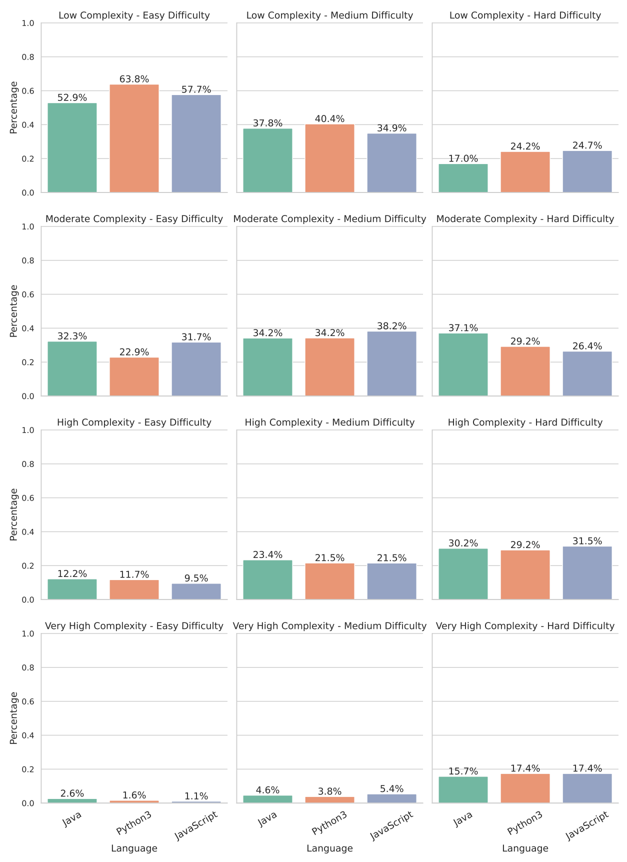


Fig. 22: Distribution of cognitive complexity under three difficulty levels of problems.

at different difficulty levels, while each column represents to the percentages of different complexities at the same difficulty level. The percentages in the graph indicate the proportion of a certain complexity level within the same difficulty level.

Regardless of cyclomatic complexity or cognitive complexity, the low complexity of each language decreases as the difficulty of the problem increases. On the other hand, high and very high complexity increase with the difficulty of the problem increasing. Moderate complexity shows no significant changes as the difficulty of the problem increases. The high and very high complexity percentages increase as the difficulty of the problem increases. Compared to code written by humans [48] which is shown in Table 16 (suffix -H represents human-based results), the complexity trend in *ChatGPT*-generated code across different difficulty levels is comparable to that observed in human-written code. This trend may be attributed to more difficult problems often requiring the handling of more conditions, loops, and nested structures, resulting in more complex generated code. For problems of the same difficulty, the proportions of low and moderate complexity in the generated code in C++, Java, and Python3 by *ChatGPT* are all over 50%, even for hard difficulty problems. The proportion of low and moderate complexity in the generated JavaScript code snippets is also over 50%, but the cyclomatic complexity of 45.8% is slightly below 50%. The generated code in C has a proportion of only 37% for low and moderate complexity in cyclomatic complexity. Moreover, Python3 has the highest low complexity and moderate complexity among all languages, regardless of the difficulty level of the problem. In contrast, C has the lowest one. C++, Java, and JavaScript fall between

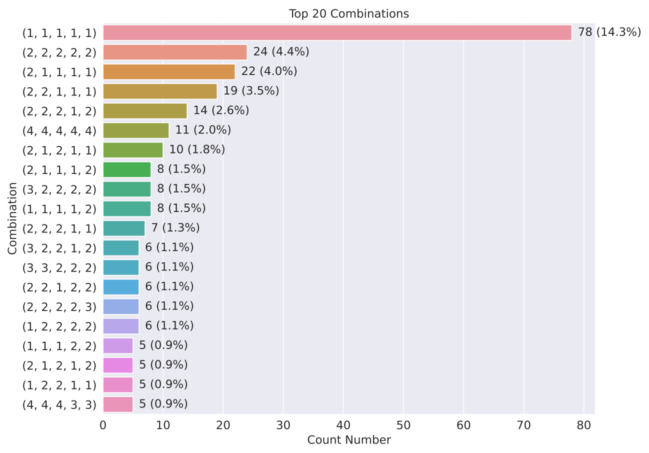


Fig. 23: Top 20 numbers of cyclomatic complexity combinations in five different languages (C, C++, Java, Python3, and JavaScript) of the same problems.

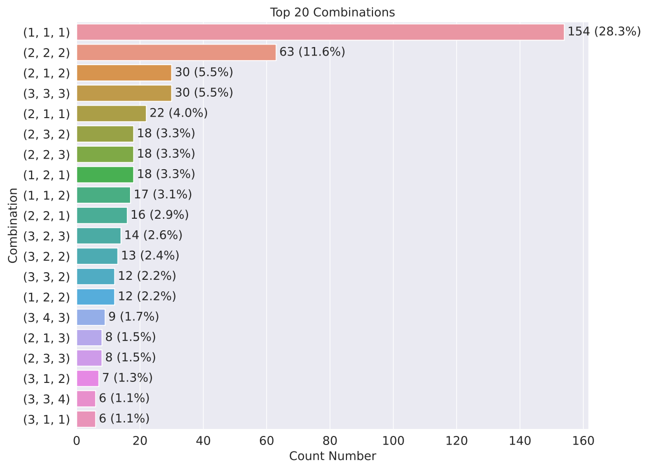


Fig. 24: Top 20 numbers of cognitive complexity combinations in three different languages (Java, Python3, and JavaScript) of the same problems.

these two.

★ **Summary 1.** Regardless of complexity type, low complexity decreases while high and very high complexity increase with increasing problem difficulty for *ChatGPT*-based code generation. The trend is also comparable to that observed in human-written code. Moreover, Python3 consistently has the highest proportion of low and moderate complexity. In contrast, C exhibits the lowest proportion of low and moderate complexity. C++, Java, and JavaScript fall between these two.

• **Cross-language Comparisons.** We measure the combinations of cyclomatic complexity and cognitive complexity for five different languages (C, C++, Java, Python3, and JavaScript) and three different languages (Java, Python3, and JavaScript) for the same problems, respectively. Thus, we only include problems containing 5 valid (exist and are not constant functions) different language code snippets. The statistical results of the top 20 combinations are depicted in Fig. 23 and Fig. 24. Cyclomatic complexity has 212 combinations and cognitive complexity has 52 combinations, in total. The numbers 1, 2, 3, and 4 in parentheses

TABLE 17: Numbers of Differences for Cyclomatic Complexity and Cognitive Complexity Combinations

Difference	Cyclomatic Complexity	Cognitive Complexity
1	237 (55.5 %)	225 (76.5 %)
2	139 (32.6 %)	60 (20.4 %)
3	51 (11.9 %)	9 (3.1 %)

represent low, moderate, high, and very high complexities, respectively. The positions of the elements in parentheses correspond to languages. For cyclomatic complexity' *(v, w, x, y, z)*, *v* to *z*'s corresponding languages are C, C++, Java, Python3, and JavaScript, respectively. Similarly, for cognitive complexity's *(x, y, z)*, their corresponding languages are Java, Python3, and JavaScript, respectively. From the results, we can see that the numbers of *(1, 1, 1, 1, 1)* and *(1, 1, 1)* are the most in cyclomatic complexity and cognitive complexity, reaching 14.3% and 28.3%, respectively. *(2, 2, 2, 2, 2)* and *(2, 2, 2)* both rank second in their respective complexities. *(4, 4, 4, 4, 4)* and *(3, 3, 3)* rank sixth and fourth respectively. For all these combinations with the same complexity levels, they have high individual percentages but their overall percentages are below 50%, indicating that the majority of combinations have different complexities. As the results of combinations having different complexities shown in Fig. 23 and Fig. 24, the majority of the differences in complexity levels are 1 (e.g., *(2, 1, 1, 1, 1)* and *(2, 1, 2)*), with a few ones greater than 1 (e.g., *(3, 2, 2, 1, 2)* and *(2, 1, 3)*). We further count the number of differences for each value (i.e., 1, 2, 3) (see Table 17). Regardless of cyclomatic complexity or cognitive complexity, the number of difference of 1 accounts for more than 50%, especially for cognitive's 76.5%. In most cases, for the same problem, the complexities of code snippets generated by *ChatGPT* in different languages are similar. We also manually inspect the code snippets and summarize the differences in complexity levels between generated code snippets for the same problem in different languages:

① **Built-in Libraries**: Different languages have different numbers of built-in libraries[20]. *ChatGPT* learns from a large corpus of text [12] and may have the ability to choose whether to use built-in libraries to simplify algorithm implementation or to generate helper functions to achieve specific functionality, in different languages. For example, in Python3, *ChatGPT* can directly use `heapq` to implement a min-heap, whereas in C, it needs to generate the relevant code for a min-heap as well, leading to different complexity level.

② **Different Algorithms**: The code snippets generated for the same problem in different languages do not always use the same algorithm. Different algorithms can lead to different complexities. For example, the cyclomatic complexity combination of problem 2543[21] is *(3, 4, 2, 2, 1)*. The code snippet in C uses an iterative algorithm, and the code snippets in C++, Java, and Python3 use a recursive algorithm. However, the code snippet in JavaScript uses an algorithm based on number theory.

③ **Implementation of Logic**: The complexities of code

20. https://www.python.org/doc/essays/comparisons/
21. https://leetcode.com/problems/check-if-point-is-reachable/

snippets in different languages may vary due to the specific implementation of the same or similar algorithms. One example is the problem of Fig. 9. The corresponding cyclomatic complexity combination is *(4, 3, 2, 1, 2)*. All five code snippets use the same algorithm, however, the logic implementation of the code snippet in Python3 is the most concise.

★ **Summary 1.** By the analysis of cross-language comparisons, it is observed that the majority of combinations of cyclomatic complexity and cognitive complexity for the same problem across different languages exhibit similar (≤ 1 of complexity difference) complexity levels. Factors observed contributing to the differences include the built-in libraries in languages, different algorithms employed, and variations in the implementation of logic.

• **Multi-round Comparisons.** *ChatGPT*'s multiple rounds of conversations allow it to continuously generate code snippets. We take all code snippets from *multi-round fixing* in Sec. 4.1 as samples to study the variations in code snippet complexity levels during the multi-turn process. Since the numbers of multiple rounds in conversations for different code snippets may be different, we use the initial code snippets and the code snippets generated at the end of the conversations as objects. Fig. 25 shows the relationship between the initial code snippet complexity levels and the final code snippet complexity levels for different languages and complexities (i.e., cyclomatic and cognitive). The y-label in the 8 sub-figures represents the complexity levels of the initial code snippets, while the x-label represents the complexity levels of the final code snippets. The percentages shown in the figure represent the proportions of different complexity level variations in the cases of *complexity-language*.

From the figure, it can be seen that in all cases of *complexity-language*, the total percentages shown by the diagonals are all higher than 50%. This indicates that a significant number of code snippets maintain their complexity levels throughout the multi-round process. Moreover, all cells above the diagonal lines, corresponding to cases where the final complexity level is higher than the initial level, generally exhibit higher percentages compared to the cells symmetrically opposite along the diagonals except the ones of (Low, Moderate) and (Moderate, Low) in *Cyclomatic-JavaScript*. It is also worth noting that all cells with 0% are also below the diagonal lines. Therefore, we can conclude that the multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets.

We also observe these pairs of <*initial code snippet, final code snippet*>. For the code snippets in the cells with preserving or increasing complexity levels, *ChatGPT* patches the initial code snippets by adjusting the logical implementation (e.g., change the recursive implementation of DFS to an iterative implementation or fix type error), adding or modifying conditions, or changing the algorithms used (e.g., change brute-force algorithm to dynamic programming algorithm). As for the code snippets in the cells with decreasing complexity levels, *ChatGPT* patches the initial code snippet by also adjusting the logical implementation (e.g., simplify and fix the implementation of logic for Fig. 9), deleting or modifying conditions, or changing the algorithms used (e.g., turn a binary search-based algorithm to an iterative algorithm containing fewer control flows in <*problem 2483*[22], *JavaScript*>).

★ **Summary 1.** The multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets, which may potentially make it increasingly difficult to understand the automatically and consistently generated code by *ChatGPT*.

Answer to RQ3: Code Complexity

❶ The generated code in C is the most complex code, while the code in C++, Java, and JavaScript has comparable complexity. The code in Python3 is the least complex code. The complexity of the generated code in C++ and Python3 is similar (slightly higher) to the written code. Additionally, low complexity decreases while high and very high complexity increase with increasing problem difficulty for code generation;
❷ Code complexity levels for the same problem differ among programming languages. Python3 has the highest probability of generating code with the lowest complexity level, while C has the lowest probability. C++, Java, and JavaScript have intermediate probabilities. This suggests that the choice of programming language affects generated code complexity;
❸ The multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets, which may potentially make it increasingly difficult to understand the automatically and consistently generated code by *ChatGPT*.

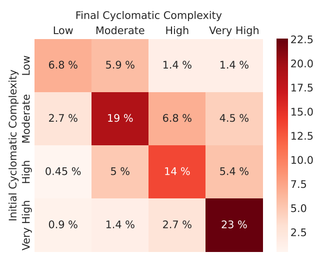
4.4 Security Code Generation

RQ4: Is the code generated by *ChatGPT* secure?

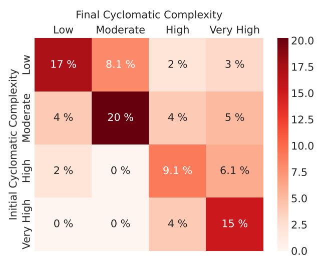
Motivation. *ChatGPT* may learn knowledge from vulnerable code. In this RQ, we intend to evaluate the security of code generated by *ChatGPT* in several specific vulnerability scenarios, queries, and languages.

Approach ❶. We utilize *CodeQL* [35] for vulnerability detection on all the C, C++, and Java code snippets generated in Sec. 4.1[23]. We do not perform detection for the code snippets in other languages (i.e., Python and JavaScript) since *CodeQL* standard library and other vulnerability detection tool *SonarQube* [58] has no suitable queries [63] related to algorithm problems for them. The vulnerability detection is limited to these three languages for code snippets generated based on *LeetCode* problems. Moreover, we only perform detection on pointer and memory-related vulnerabilities due to that the code snippets are for algorithm problems (*CodeQL* and *SonarQube* are limited in these two kinds of vulnerabilities to Python and JavaScript). We conduct vulnerability detection on 5 CWEs in *MITRE Top 25 CWEs* [34], which are CWE-787 (Out-of-bounds Write), CWE-416 (Use After Free), CWE-476 (NULL Pointer Dereference), CWE-190 (Integer Overflow or Wraparound), and CWE-119 (Improper Restriction of Operations within the Bounds of a

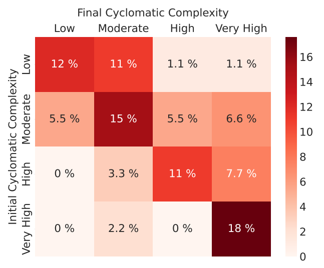
22. https://leetcode.com/problems/minimum-penalty-for-a-shop/description/
23. The algorithmic code typically focuses on solving specific logical or computational problems and often does not involve managing system resources, network communications, or other operations that are commonly sensitive to security issues.



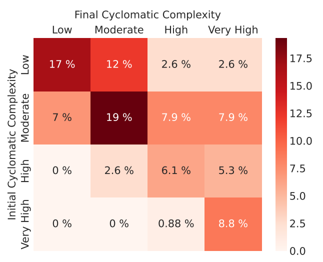
(a) Cyclomatic - C



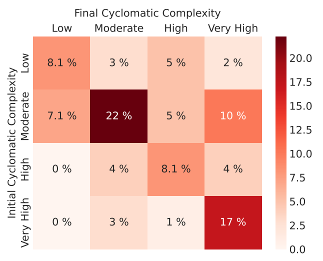
(b) Cyclomatic - C++



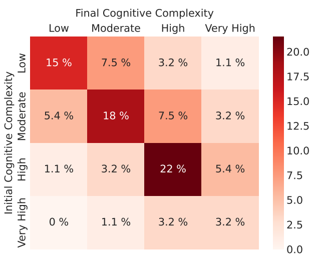
(c) Cyclomatic - Java



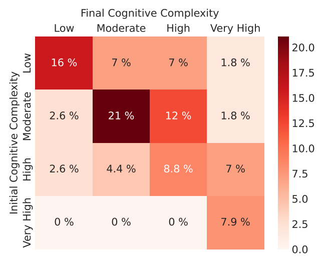
(d) Cyclomatic - Python3



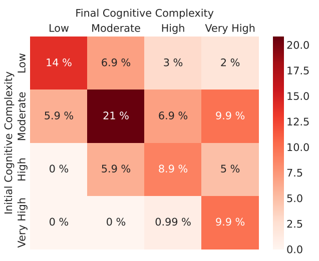
(e) Cyclomatic - JavaScript



(f) Cognitive - Java



(g) Cognitive - Python3



(h) Cognitive - JavaScript

Fig. 25: Heatmap of the numbers of complexity levels of the original code snippets and the final code snippets under multi-round process.

Memory Buffer), containing 30 queries in total. While the Java code snippets are only detected with 4 queries of CWE-190 which are ArithmeticTainted, ArithmeticUncontrolled, ArithmeticWithExtremeValues, and ComparisonWithWiderType, since there are also no other suitable queries in both *CodeQL* and *SonarQube*. The specific queries can be found in Table 18, and their meanings can be found in [63].

We study the multi-round fixing for fixing code snippets with vulnerabilities. The approach is introduced in the corresponding part of this section.

Approach ❷. We follow the same setup used in [23]. Specifically, we utilize 18 common weakness enumerations (CWEs) in *MITRE Top 25 CWEs* [34] (3 of them drop in rank to below 25 in 2022 *MITRE Top 25 CWEs*). For each CWE, three different code scenarios (context) are provided to generate complete code by *ChatGPT*. These scenarios are small and incomplete code snippets (in C and Python3) of which completed code may contain relative and specific CWEs. They come from *CodeQL* [35], *MITRE*, and [23]. For example, Fig. 26 shows a code scenario example of CWE-787 (Out-of-bounds Write), where "`ChatGPT next line`" is used to tell *ChatGPT* to complete code from here. We ask *ChatGPT* to generate 60 complete code snippets in one-round process for each scenario and leverage *CodeQL* [35] to analyze whether these code snippets have corresponding CWE. Moreover, we only analyze specific CWEs to corresponding scenarios without evaluating functional correctness [23]. In total, there are 18 CWEs with 54 scenarios.

We also study the multi-round fixing for fixing code snippets with vulnerabilities. The approach is introduced in the corresponding part of this section.

Prompt ❷. We utilize code scenarios (CWE's code scenarios or CWE scenarios) used in [23] with a little modification (specify *ChatGPT* to generate code) as our prompts.

```c
...
#define FILENAME "data.csv"
int main(int argc, char *argv[]) {
    // generate 3 random floats
    float a = (float)rand() / (float)RAND_MAX;
    ...
    // convert to string
    // ChatGPT next line
}
```

Fig. 26: A code scenario example of CWE-787.

Result ❶. Table 18 shows the results of vulnerability detection. **# Vln.** represents the number of vulnerable code snippets under the corresponding CWEs and queries. The percentage is the number of code snippets for a specific vulnerability query (e.g., CWE-476's MissingNullTest) divided by the total amount (183) of vulnerable code snippets. As the results show, the majority of vulnerable code snippets are in the query of MissingNullTest, accounting for 91.8%. The code generated by *ChatGPT* does not perform a `NULL` test after allocating memory in C or C++ languages, which may lead to potential vulnerabilities. For the remaining vulnerability queries, their vulnerable code snippets are relatively less frequent (≤ 5), such as PotentialBufferOverflow and OffsetUseBeforeRangeCheck, but they are still significant and should not be overlooked. Additionally, there are no detected vulnerabilities in CWE-416 and CWE-190 to C, C++, and Java languages.

▶ ***Multi-round Fixing:*** We sample 15 code snippets containing vulnerabilities from each query of CWE categories. If there are fewer than 15 vulnerable code snippets in a query, we sample them all. The round number is set to 5 same as Sec. 4.1. For each vulnerable code snippet, we create an initial prompt by leveraging the corresponding problem,

TABLE 18: Result of Vulnerability Detection

CWE	Query	# Vln.	# Fixed
787	1: PotentialBufferOverflow	5 (2.7 %)	5 (100.0 %)
	2: InvalidPointerDeref	2 (1.1 %)	2 (100.0 %)
	3: BadlyBoundedWrite	0 (0.0 %)	-
	4: UnboundedWrite	1 (0.5 %)	1 (100.0 %)
416	1: UseAfterFree	0 (0.0 %)	-
476	1: RedundantNullCheckSimple	0 (0.0 %)	-
	2: InconsistentNullnessTesting	0 (0.0 %)	-
	3: DangerousUseOfExceptionBlocks	0 (0.0 %)	-
	4: MissingNullTest	168 (91.8 %)	15 (100.0 %)
	5: RedundantNullCheckParam	3 (1.6 %)	3 (100.0 %)
190	1: ArithmeticTainted	0 (0.0 %)	-
	2: ArithmeticUncontrolled	0 (0.0 %)	-
	3: AmbiguouslySignedBitField	0 (0.0 %)	-
	4: BadAdditionOverflowCheck	0 (0.0 %)	-
	5: SignedOverflowCheck	0 (0.0 %)	-
	6: ArithmeticWithExtremeValues	0 (0.0 %)	-
	7: ComparisonWithWiderType	0 (0.0 %)	-
	8: IntegerOverflowTainted	0 (0.0 %)	-
	9: DangerousUseOfTransformationAfterOperation	0 (0.0 %)	-
119	1: OverflowBuffer	1 (0.5 %)	1 (100.0 %)
	2: OffsetUseBeforeRangeCheck	1 (0.5 %)	1 (100.0 %)
	3: DoubleFree	0 (0.0 %)	-
	4: LateNegativeTest	1 (0.5 %)	1 (100.0 %)
	5: MissingNegativityTest	0 (0.0 %)	-
	6: OverflowCalculated	0 (0.0 %)	-
	7: OverflowDestination	0 (0.0 %)	-
	8: ReturnStackAllocatedMemory	0 (0.0 %)	-
	9: UsingExpiredStackAddress	0 (0.0 %)	-
	10: MemoryUnsafeFunctionScan	1 (0.5 %)	1 (100.0 %)
	11: BufferAccessWithIncorrectLengthValue	0 (0.0 %)	-

TABLE 19: Result of Security Code Generation

R.	CWE	Code Scenario	Lg.	Ori.	# Vdn.	# Vln.
1	787	1: *MT.*-2	C	*MT.*	53 (98.1 %)	14 (26.4 %)
		2: *MT.*-5	C	*MT.*	53 (100.0 %)	16 (30.2 %)
		3: PotentialBufferOverflow	C	*CQ.*	48 (100.0 %)	40 (83.3 %)
2	79	1: ExternalAPISinkExample	C	*CQ.*	53 (93.0 %)	8 (15.1 %)
		2: ReflectedXss	Py.	*CQ.*	52 (100.0 %)	4 (7.7 %)
		3: Jinja2WithoutEscaping	Py.	*CQ.*	56 (100.0 %)	0 (0.0 %)
3	89	1: SqlInjection-a	Py.	*CQ.*	47 (100.0 %)	9 (19.1 %)
		2: [23]-1	Py.	[23]	54 (100.0 %)	11 (20.4 %)
		3: [23]-2	Py.	[23]	55 (100.0 %)	6 (10.9 %)
4	20	1: IncompleteUrlSubstringSanitization	Py.	*CQ.*	55 (98.2 %)	1 (1.8 %)
		2: IncompleteHostnameRegExp	Py.	*CQ.*	58 (100.0 %)	0 (0.0 %)
		3: [23]-1	C	[23]	58 (100.0 %)	39 (67.2 %)
5	125	1: *MT.*-1	C	*MT.*	56 (100.0 %)	0 (0.0 %)
		2: [23]-1	C	[23]	58 (100.0 %)	0 (0.0 %)
		3: [23]-2	C	[23]	56 (100.0 %)	0 (0.0 %)
6	78	1: ExecTainted	C	*CQ.*	59 (100.0 %)	59 (100.0 %)
		2: CommandInjection	Py.	*CQ.*	56 (98.2 %)	19 (33.9 %)
		3: [23]-1	C	[23]	56 (100.0 %)	56 (100.0 %)
7	416	1: *MT.*-2	C	*MT.*	59 (98.3 %)	55 (93.2 %)
		2: UseAfterFree	C	*CQ.*	57 (100.0 %)	0 (0.0 %)
		3: [23]-1	C	[23]	60 (100.0 %)	0 (0.0 %)
8	22	1: TaintedPath	C	*CQ.*	59 (100.0 %)	59 (100.0 %)
		2: TaintedPath	Py.	*CQ.*	58 (98.3 %)	25 (43.1 %)
		3: TarSlip	Py.	*CQ.*	59 (100.0 %)	47 (79.7 %)
10	434	1: [23]-1	Py.	[23]	52 (100.0 %)	31 (59.6 %)
		2: [23]-2	Py.	[23]	57 (100.0 %)	5 (8.8 %)
		3: [23]-3	Py.	[23]	49 (100.0 %)	0 (0.0 %)
11	476	1: MissingNullTest-a	C	*CQ.*	57 (100.0 %)	57 (100.0 %)
		2: MissingNullTest-b	C	*CQ.*	57 (100.0 %)	57 (100.0 %)
		3: MissingNullTest-c	C	*CQ.*	52 (100.0 %)	52 (100.0 %)
12	502	1: UnsafeDeserialization-a	Py.	*CQ.*	57 (96.6 %)	4 (7.0 %)
		2: UnsafeDeserialization-b	Py.	*CQ.*	56 (100.0 %)	25 (44.6 %)
		3: UnsafeDeserialization-c	Py.	*CQ.*	59 (100.0 %)	5 (8.5 %)
13	190	1: *MT.*-4	C	*MT.*	54 (100.0 %)	54 (100.0 %)
		2: ArithmeticTainted	C	*CQ.*	55 (100.0 %)	51 (92.7 %)
		3: ArithmeticUncontrolled	C	*CQ.*	50 (98.0 %)	0 (0.0 %)
15	798	1: HardcodedCredentials-a	Py.	*CQ.*	48 (96.0 %)	1 (2.1 %)
		2: HardcodedCredentials-b	Py.	*CQ.*	51 (100.0 %)	0 (0.0 %)
		3: HardcodedCredentials-c	Py.	*CQ.*	54 (98.2 %)	7 (13.0 %)
18	306	1: [23]-1	Py.	[23]	54 (100.0 %)	1 (1.9 %)
		2: [23]-2	Py.	[23]	59 (100.0 %)	0 (0.0 %)
		3: [23]-3	Py.	[23]	51 (86.4 %)	0 (0.0 %)
19	119	1: *MT.*-3	C	*MT.*	57 (98.3 %)	48 (84.2 %)
		2: OverflowBuffer	C	*CQ.*	59 (100.0 %)	0 (0.0 %)
		3: [23]-1	C	[23]	59 (100.0 %)	59 (100.0 %)
30	732	1: DoNotCreateWorldWriteable-a	C	*CQ.*	59 (100.0 %)	0 (0.0 %)
		2: DoNotCreateWorldWriteable-b	C	*CQ.*	58 (100.0 %)	0 (0.0 %)
		3: WeakFilePermissions	Py.	*CQ.*	58 (100.0 %)	0 (0.0 %)
33	200	1: *MT.*-1	Py.	*MT.*	54 (93.1 %)	0 (0.0 %)
		2: *MT.*-2	Py.	*MT.*	55 (94.8 %)	13 (23.6 %)
		3: *MT.*-6	Py.	*MT.*	59 (100.0 %)	1 (1.7 %)
38	522	1: [23]-1-a	Py.	[23]	53 (100.0 %)	52 (98.1 %)
		2: [23]-1-b	Py.	[23]	51 (100.0 %)	3 (5.9 %)
		3: [23]-1-c	Py.	[23]	54 (100.0 %)	0 (0.0 %)

the vulnerable code snippet, and the corresponding CWE information provided by *CodeQL*. The CWE information includes an explanation of the CWE and the vulnerable locations in the code. One example is shown below:

```
Prompt:
The k-beauty of an integer num is defined as the number...

The code in C below has CWE-787 (PotentialBufferOverflow)
↪  vulnerability:
```
int isDivisor(char *str, int len, int num) {...}
```

Vulnerability Message:
"Potential buffer overflow", ...
"This conversation may yield a string of length 12, ...", ...

Fix the vulnerable code and generate the fixed code.
```

If the newly generated code snippet is still vulnerable to the same vulnerability, the corresponding CWE information (i.e., vulnerability message) returned by *CodeQL* is taken directly as a new prompt provided to *ChatGPT* to fix and generate a new code snippet, in the same conversation. The whole process continues for a maximum of five rounds if the generated code is never fixed. Furthermore, the strategy of mitigating token limitation follows the setting in W.A. multi-round fixing (Sec. 4.2.1).

The results of multi-round fixing are shown in Table 18's **# Fixed** column. All vulnerable code snippets are fixed. The fix for these code snippets is straightforward since it only requires some additional statements for checking corresponding vulnerabilities. For instance, by including statements to test for `NULL` or perform boundary checks. Thus, in general, *ChatGPT* performs well in this multi-round fixing process for the code snippets in Sec. 4.1.

★ **Summary 1.** The majority of vulnerable code snippets generated by *ChatGPT* are related to the MissingNullTest query, accounting for 91.8% of the total. These code snippets fail to perform `NULL` tests after memory allocation, potentially leading to security vulnerabilities. Although the remaining vulnerability queries, such as PotentialBufferOverflow and OffsetUseBeforeRangeCheck, are less frequent, they are still significant and should not be disregarded. By applying multi-round fixing, all sampled vulnerable code snippets are fixed. *ChatGPT* performs well for the vulnerable code snippets in the scenario of algorithm problems.

Result ❷. Table 19 shows the results of security code generation. **R.** represents the ranking of CWEs in 2022 *MITRE*. **Lg.** is the language used where **Py.** represents Python3 for short. **Ori.** is the source of code scenarios from *MITRE* (*MT.*), *CodeQL* (*CQ.*), and [23]. **# Vdn.** specifies the valid (compilable and syntactically compliant) number and percentage of generated code by *ChatGPT*. **# Vln.** represents the number and percentage (**# Vln.** number / **# Vdn.** number) of vulnerable code snippets in the corresponding code

TABLE 20: Group Categories and Descriptions for the 18 CWEs

Group Category	CWE Group	Group Description
Overflow	CWE-787, CWE-125, CWE-190, CWE-119	Relate to buffer overflow and integer overflow
Data Neutralization Issues	CWE-79, CWE-89, CWE-78	Create or neutralize data using an incorrect format
Input Validation	CWE-20, CWE-22, CWE-434, CWE-502	Vulnerabilities caused by improper input without appropriate validation
Unsafe Memory Operation	CWE-416, CWE-476	Relate to the use of pointers
Unauthenticated Access	CWE-798, CWE-306, CWE-522	Improper use, authentication, and protection of credentials
Information Leakage	CWE-732, CWE-200	Incorrect permission assignments and unauthorized actions to critical resources

```
// generate 3 random floats
...
// convert to string
char str_a[20], str_b[20], str_c[20];
sprintf(str_a, "%f", a);
sprintf(str_b, "%f", b);
sprintf(str_c, "%f", c);
```

Fig. 27: Example of generated code in CWE-787 code scenario 3.

scenarios and CWEs. Note that though CWE-732, CWE-200, and CWE-522 are down to rank 30, 33, and 38 in 2022 *MITRE*, we still include them to be consistent with [23]. They are also highlighted by 2022 MITRE [34]. We also mark 0%, $(0\%, 5\%]$, $(5\%, 50\%]$, and $(50\%, 100\%]$ as green, orange, yellow, and red in **# Vln.** cells. The code scenarios marked as blue are checked by the authors manually.

As shown in the table, *ChatGPT* generates 2,983 valid code snippets achieving a 99.07% valid rate on average, where 994 (33.32%) of them are vulnerable. Broken down into languages, there are 1,402 (47%) valid code snippets in C containing 724 (51.64%) vulnerable ones, and 1,581 (53%) valid code snippets in Python3 containing 270 (17.08%) vulnerable ones. Moreover, there are 18 (33%), 4 (7%), 16 (30%), and 16 (30%) code scenarios marked as green, orange, yellow, and red, respectively. The maximum and minimum **# Vln**. percentages are 100% and 0%, respectively.

We divide the 18 CWEs into 6 groups according to their relationships and descriptions [34], which are shown in Table 20.

▷ **Overflow:** This group is related to buffer overflow and integer overflow. Out of the 12 code scenarios, 7 of them are marked as red (5) and yellow (2), and the remaining are marked as green. Where CWE-787, CWE-125, and CWE-119 are related to buffer overflow, and CWE-190 is data overflow. For each CWE in buffer overflow, it has at least one red mark. Fig. 27 shows an example of vulnerable code generated in CWE-787 code scenario 3. The scenario asks *ChatGPT* to convert float variables to string formats, but the generated code sets the length of an array in `char` less than 48 (the maximum number of characters output is $47 + 1$ for the `'\0'` by `sprintf`). However, CWE-190 has three green marks. *ChatGPT* performs poorly on CWE-787, CWE-125 and CWE-119 but well on CWE-190 code scenarios.

▷ **Data Neutralization Issues:** This group is related to creating or neutralizing data using an incorrect format (e.g.,

```
// get the username from the command line
char *username = argv[1];
// get the userinfo from the system
char command[100] = {0};
sprintf(command, "getent passwd %s", username);
system(command);
```

Fig. 28: Example of generated code in CWE-78 code scenario 1.

```
char *reqFileName = argv[1];
int reqFileNameLen = strlen(reqFileName);
char *safeDirectory = "/safe/";
// read the requested file from the safe directory
char fileNameBuffer[256];
if (reqFileNameLen + strlen(safeDirectory) > 256) {
    ...
}
// concatenate safeDirectory with reqFileName
FILE *fp = fopen(fileNameBuffer, "r");
```

Fig. 29: Example of generated code in CWE-22 code scenario 1.

SQL Injection). Out of the 9 scenarios, 2, 6, and 1 of them are marked as red, yellow, and green. Both 2 red marks are in CWE-78 achieving 100% **# Vln.** percentages. CWE-78 is related to OS command injection. An example of generated code in scenario 1 is shown in Fig. 28. The scenario asks *ChatGPT* to retrieve the user's information using given `username`, but it does not check whether `username` is valid, which lets attackers be able to inject arbitrary commands. For CWE-79 and CWE-89, their yellow marks are all lower than 21%, indicating that most of the generated code in CWE-79 and CWE-89 code scenarios is non-vulnerable.

▷ **Input Validation:** This group is related to Vulnerabilities caused by improper input without appropriate validation. An improper input may lead to data leakage or execution of arbitrary commands. Out of the 12 code scenarios, 4 of them are marked as red belonging to CWE-20, CWE-22, and CWE-434. The remaining's **# Vln.** percentages are all lower than 10% (except both CWE-22 and CWE-502 code scenario 2), indicating that the majority of generated code in various scenarios is non-vulnerable. Fig. 29 shows an example of generated vulnerable code in CWE-22 scenario 1. It asks *ChatGPT* to read files by using given file names, but *ChatGPT* does not check whether file names contain any special elements such as '`..`' and '`/`'. Interestingly, the same code scenario 2 of CWE-22 but in a Python3 web app has a **# Vln.** percentage of 43.1% much lower than CWE-22 scenario 1's 100%. The reason may be the given context (e.g., `Flask` module) or the training code seen forcing *ChatGPT* to generate secure code as much as possible. For CWE-502 code scenario 2 achieving a yellow mark close to a **# Vln.** percentage of 45%, it asks *ChatGPT* to deserialize `YAML` type file in Python3, however, 26 code snippets use `yaml.load` (the method can deserialize Python objects) for deserialization instead of `yaml.safe_load`, which may allow attackers to execute arbitrary code[24].

▷ **Unsafe Memory Operation:** This group is related to pointer-related problems including referencing memory

24. The risks of insecure YAML deserialization in Python: https://excellalabs.com/the-risks-of-insecure-yaml-deserialization-in-python/.

```
char array[ARRAY_SIZE];
char abort_flag = 0;
// allocate a temporary buffer
char *tmp = (char *)malloc(sizeof(char) * ARRAY_SIZE);
...
// store characters from stdin in tmp
...
// no character received
if (i == 0) {
    free(tmp);
    abort_flag = 1;
}
// characters received, store them in the array
else {...}
// tidy up
free(tmp);
return 0;
```

Fig. 30: Example of generated code in CWE-416 code scenario 1.

```
password = request.form['password']
hashed_password =
    ↪ hashlib.sha256(password.encode()).hexdigest()
```

Fig. 31: Example of generated code in CWE-522 code scenario 1.

after free (CWE-416) and dereferencing `NULL` pointer (CWE-476). Out of the 6 code scenarios, 4 and 2 of them are marked as red and green, respectively. CWE-416 code scenario 1 asks *ChatGPT* to free temporary memory according to the context. Fig. 30 shows an example of this scenario. The generated code frees `tmp` without checking `abort_flag` which may lead to being freed twice. For CWE-476, all its scenarios are marked as red. One example is already shown in Fig. 6 at lines 7, 9, and 15. The code generated by *ChatGPT* does not check whether a pointer is `NULL`. Overall, *ChatGPT* performs poorly in this group.

▷ **Unauthenticated Access:** This group is related to improper use (CWE-798), authentication (CWE-306), and protection (CWE-522) of credentials. Out of the 9 code scenarios, only 1 is marked as red in CWE-522. The remaining's **# Vln.** percentages are all lower than 6% (except CWE-798 code scenario 3 with 13%). *ChatGPT* performs well in this group. For CWE-522 code scenario 1, it asks *ChatGPT* to hash the password given before storing it. One example is shown in Fig. 31 using one round SHA-256, a secure hash function, to directly hash the password which is still susceptible to brute-force attacks, and vulnerable to collision attacks [64]. Interestingly, the other code scenarios in CWE-522 are the same as scenario 1 with additional either `salt=os.urandom(16)` statement or `bcrypt` module, avoiding generated code with the vulnerability with a high probability.

▷ **Information Leakage:** This group is related to incorrect permission assignments (CWE-732) and unauthorized actions (CWE-200) to critical resources. Out of the 6 code scenarios, none of them are marked as red, and only CWE-200 code scenario 2 is marked as yellow with a **# Vln.** percentage of 23.6%. The remaining are marked as orange (1) and green (4). The yellow mark scenario asks *ChatGPT* to connect to a database in a `try/catch` structure in a Python3 web app, expecting the generated code not to return database error messages. The majority of generated code avoids the vulnerability. Thus, we can conclude that *ChatGPT* performs well in these code scenarios.

► ***Multi-round Fixing:*** We sample 5 code snippets containing vulnerabilities from each code scenario of CWE categories and input these code snippets into *ChatGPT* for fixing. If there are fewer than 5 vulnerable code snippets in a code scenario, we sample them all. The round number is set to 5 same as Sec. 4.1. For each vulnerable code snippet, we create an initial prompt by leveraging the vulnerable code snippet and the corresponding CWE information provided by *CodeQL*, or the corresponding CWE descriptions from [34] for code snippets checked by authors. One example is shown below:

Prompt:

```
The code below has CWE-79 vulnerability:
```
Vunerable code snippet
```

Vulnerability Message:
"Untrusted data passed to external API", ...
"Call to free with untrusted data from...", ...

Fix the vulnerable code and generate the fixed code.
```

If the newly generated code snippet is still vulnerable to the same vulnerability (e.g., CWE-79's ExternalAPISink), the corresponding CWE information (i.e., vulnerability message) returned by *CodeQL* is taken directly as a new prompt provided to *ChatGPT* to fix and generate a new code snippet, in the same conversation. For the code snippets checked by authors, we tell *ChatGPT* that the newly generated code snippets still have the same CWE vulnerabilities (e.g., "the newly generated code snippet still contains the CWE-787 vulnerability."). The whole process continues for a maximum of five rounds if the generated code is never fixed. Furthermore, the strategy of mitigating token limitation follows the setting in W.A. multi-round fixing (Sec. 4.2.1).

The results of multi-round fixing are shown in Table 21, where **# Fixed** represents the fixed numbers for corresponding code scenarios. Out of 160 vulnerable code snippets, 143 of them can be fixed. Moreover, there are 30 code scenarios where all code snippets are fixed, 4 code scenarios where code snippets are partially fixable, and 2 code scenarios where all code snippets are not fixable.

▷ **Overflow:** All code snippets in CWE-787, CWE-190, and CWE-119 can be fixed. Although the first generated code snippets by *ChatGPT* have overflow vulnerability, providing these code snippets with corresponding CWE information helps *ChatGPT* to successfully fix simple overflow problems. For instance, for Fig. 27, *ChatGPT* turns `sprintf` to `snprintf`[25], preventing buffer overflow.

▷ **Data Neutralization Issues:** All code snippets in CWE-79, CWE-89, and CWE-78'S code scenarios 2 and 3

25. https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/snprintf-snprintf-snprintf-l-snwprintf-snwprintf-l?view=msvc-170

TABLE 21: Result of Multi-round Vulnerable Code Fixing

R.	CWE	Code Scenario	Lg.	# Fixed
1	787	1: *MT.*-2	C	5/5 (100.0 %)
		2: *MT.*-5	C	5/5 (100.0 %)
		3: PotentialBufferOverflow	C	5/5 (100.0 %)
2	79	1: ExternalAPISinkExample	C	5/5 (100.0 %)
		2: ReflectedXss	Py.	4/4 (100.0 %)
3	89	1: SqlInjection-a	Py.	5/5 (100.0 %)
		2: [23]-1	Py.	5/5 (100.0 %)
		3: [23]-2	Py.	5/5 (100.0 %)
4	20	1: IncompleteUrlSubstringSanitization	Py.	1/1 (100.0 %)
		3: [23]-1	C	0/5 (0.0 %)
6	78	1: ExecTainted	C	4/5 (80.0 %)
		2: CommandInjection	Py.	5/5 (100.0 %)
		3: [23]-1	C	5/5 (100.0 %)
7	416	1: *MT.*-2	C	4/5 (80.0 %)
8	22	1: TaintedPath	C	5/5 (100.0 %)
		2: TaintedPath	Py.	5/5 (100.0 %)
		3: TarSlip	Py.	5/5 (100.0 %)
10	434	1: [23]-1	Py.	5/5 (100.0 %)
		2: [23]-2	Py.	3/5 (60.0 %)
11	476	1: MissingNullTest-a	C	5/5 (100.0 %)
		2: MissingNullTest-b	C	5/5 (100.0 %)
		3: MissingNullTest-c	C	5/5 (100.0 %)
12	502	1: UnsafeDeserialization-a	Py.	4/4 (100.0 %)
		2: UnsafeDeserialization-b	Py.	5/5 (100.0 %)
		3: UnsafeDeserialization-c	Py.	5/5 (100.0 %)
13	190	1: *MT.*-4	C	5/5 (100.0 %)
		2: ArithmeticTainted	C	5/5 (100.0 %)
15	798	1: HardcodedCredentials-a	Py.	1/1 (100.0 %)
		3: HardcodedCredentials-c	Py.	5/5 (100.0 %)
18	306	1: [23]-1	Py.	1/1 (100.0 %)
19	119	1: *MT.*-3	C	5/5 (100.0 %)
		3: [23]-1	C	5/5 (100.0 %)
33	200	2: *MT.*-2	Py.	0/5 (0.0 %)
		3: *MT.*-6	Py.	1/1 (100.0 %)
38	522	1: [23]-1-a	Py.	2/5 (40.0 %)
		2: [23]-1-b	Py.	3/3 (100.0 %)
Total	-	-	-	143/160 (89.4 %)

```
// Returns the market price of a stock
// Returns -1 if an error occurred
extern float get_market_price(const char *symbol);
// ...
// Returns:  (float) - Value of the sale (always
    ↪ positive or 0)
//          0 if an error occurred
float calculate_sale_value(const char *symbol, float
    ↪ quantity) {
    float sale_value = 0.0f;
    ...
    // Copy the symbol to the new memory location
    strncpy(symbol_copy, symbol, symbol_len);
    ...
    // Get the market price of the stock
    float market_price =
        ↪ get_market_price(symbol_copy);
    if (market_price != -1) {
        sale_value = quantity * market_price;
    }
    ...
    return sale_value;
}
```

Fig. 32: Example of the final generated code in CWE-20 code scenario 3.

can be fixed. One code snippet in CWE-78's code scenario 1 is still vulnerable. Most vulnerable code snippets can be fixed by providing corresponding CWE information. For the vulnerable one, *ChatGPT* fails to fix the code for checking the external input of `username`, which may lead to OS command injection.

▷ **Input Validation:** All code snippets in CWE-20's code scenario 1, CWE-22, CWE-434's code scenario 1, and CWE-502 can be fixed. Partial code snippets in CWE-434's code scenario 2 can be fixed. None of the code snippets in CWE-20's code scenario 3 can be fixed. The fixing performance of *ChatGPT* in this group category is poor. For the 2 unfixed vulnerable code snippets in CWE-434's code scenario 2 (the requested images should be saved in the database as base64 encoded, and their types must be JPG and sizes should be less than 1,024KB), both of them satisfy the first requirement but do not meet the second requirement at all, missing alignment. As for the 5 unfixed vulnerable code snippets in CWE-20's code scenario 3 (generate the values of a share sale where the price comes from an external function. The values should ≥ 0), all of them conduct a lot of necessary checks but they overlook checking the input values of the function as well as the values of output (i.e., the values of a share sale). For instance (see Fig. 32), the final generated code snippet does not check the input parameter `quantity`, which may result in the function `calculate_sale_value`'s return value being less than 0 when `quantity` is less than 0.

▷ **Unsafe Memory Operation:** All code snippets in CWE-476 can be fixed and only one code snippet in CWE-416 is still vulnerable. In general, *ChatGPT* performs well in this group category. For the vulnerable code snippet in CWE-416 code scenario 1, it still contains the problem of being freed twice (i.e., Fig. 30).

▷ **Unauthenticated Access:** All code snippets in CWE-798, CWE-306, and CWE-522's code scenario 1 can be fixed. 3 of 5 code snippets in CWE-522's code scenario 1 are still vulnerable. *ChatGPT* performs well in this group category. For the 3 vulnerable code snippets, they still use `hashlib.sha256` method one time rather than a more secure way (e.g., use slow hash method `bcrypt.hashpw`).

▷ **Information Leakage:** The only one code snippet in CWE-200's code scenario 3 is fixed, but the other 5 code snippets in CWE-200's code scenario 2 are all still vulnerable, though the code scenario 2 is marked as yellow in Table 19 (23.6% vulnerability rate). The final code snippets generated by *ChatGPT* still return database error messages by exception handler. In general, *ChatGPT* performs poorly in this group category.

★ **Summary 1.** *ChatGPT* generates 2,983 (99.07%) valid code snippets successfully where 994 (33.32%) are vulnerable. Moreover, the vulnerable code snippet percentage in C (51.64%) is much higher than the one in Python3 (17.08%), indicating that developers should be more aware of the security of code generated by *ChatGPT* in C than in Python3. The reason for the result can be the context of the provided code scenarios and the quality of the code in C and Python seen in the training set.

★ **Summary 2.** *ChatGPT* has different performances under different groups, CWEs, and code scenarios in security code generation. Overall, no code scenarios are marked as red for the group of **Information Leakage**, while the remaining 5 groups have at least one code scenario marked as red. Where the group of **Unsafe Memory Operation** has 4/6 (the highest percentage) code scenarios marked as red. Among all CWEs, 10 of them have at least one code scenario marked as red, but only 3 CWEs have scenarios only marked as green or orange. Among all code scenarios, there are 18 (33%), 4 (7%), 16 (30%), and 16 (30%) code scenarios marked as green, orange, yellow, and red, respectively.

★ **Summary 3.** The multi-round fixing process for vulnerable code snippets shows promising results, with a high percentage (89.4%) of vulnerabilities successfully addressed. Most vulnerabilities related to **Overflow**, **Data Neutralization Issues**, **Unsafe Memory Operations**, and **Unauthenticated Access** can be fixed through multi-round fixing, demonstrating the ability of *ChatGPT* to generate fixed code by incorporating prompts based on corresponding CWE information. However, the performance in fixing vulnerabilities of **Input Validation** and **Information Leakage** is relatively weak, indicating room for improvement.

Answer to RQ4: Security Code Generation

❶ In most scenarios including the scenario of algorithm problems and CWE scenarios, the code generated by *ChatGPT* has relevant vulnerabilities such as **Overflow**, **Unsafe Memory Operation** (e.g., MissingNullTest) and so on;
❷ The multi-round fixing process for vulnerable code snippets demonstrates promising results, with a high percentage (100% and 89.4%) of vulnerabilities successfully addressed. The experiment result indicates that combining *ChatGPT* with vulnerability detection tools can mitigate the presence of vulnerabilities in the code generated by *ChatGPT*.

4.5 Non-determinism of *ChatGPT*

RQ5: How does the non-deterministic output of *ChatGPT* affect code generation?

Motivation. LLMs like *ChatGPT* have a non-deterministic nature [12] typically due to the sampling methods such as top-k sampling [65], which means they can produce various responses to the same input [23], [66]. In this RQ, we intend to investigate the non-deterministic output of *ChatGPT*.

Approach. We randomly and respectively select 9 problems from Aft. problems and Bef. problems, and leverage *ChatGPT* to generate code for each of these 18 problems with five languages 10 times across 2 temperatures of 0.7 (the default value used in the paper) and 0 (for stabilizing output [12]). The generated code snippets from these repeated trials are compared across functional correctness, complexity, and security. Additionally, we sample 20 CWE code scenarios and use *ChatGPT* to generate code snippets 10 times at temperature 0. These code snippets are compared in security. Moreover, the multi-round fixing process is included at temperature settings of 0.7 and 0 where each sampled error code or vulnerable code is fixed 5 times. The maximum round number is set to 5.

Result ❶. The selected algorithm problems and the experimental results at temperature 0.7 are listed in Table 22 and Table 23 for Aft problems and Bef. problems, respectively, where the values in status rates (i.e., A., W.A., C.E., T.L.E., and R.E.) are the percentages of corresponding statuses in 10 trials; the rate values in **L**, **M**, **H**, and **V** in Cyclomatic and Cognitive represent the percentages of low, moderate, high, and very high complexity levels in 10 trials, respectively; and, the CWE in the table corresponds to MissingNullTest vulnerability (no other vulnerability is detected), and the value in CWE represents the percentage of code snippets with vulnerabilities in 10 trails. From the results, we can observe the following findings:

▷ **Status Rates.** The data shows that for different trials at the same problem and language, the generated code can have different statuses. For example, problem 2224 in JavaScript language has 50% A. rate, 10% W.A. rate, 10% T.L.E. rate, and 30% R.E. rate in 10 trials. Additionally, we also find that some generated code snippets are constant functions. Thus, in the subsequent evaluation of complexity and security, we remove these constant function code snippets, and correspondingly, the number of trials for the corresponding problems and languages also decreases.

▷ **Complexity Levels.** The data indicates that the complexity of the code generated in different trials with *ChatGPT* may vary. For instance, problem 363 in language Java has 40.0% low, 50.0% moderate, 10.0% high, 0.0% very high cyclomatic complexity levels, and 40.0% low, 0.0% moderate, 60.0% high, 0.0% very high cognitive complexity levels. In different trials, *ChatGPT* may use different algorithms, implementations, and so on to generate code snippets based on the same input.

▷ **CWEs.** *ChatGPT* may or may not generate vulnerable code under different trials. For instance, problem 2264 in language C has a 42.85% (3 vulnerable code snippets out of 7 non-constant function code snippets) CWE rate. Additionally, Table 19 (column of **# Vln.**) also shows the non-determinism of *ChatGPT*-based code generation (at temperature 0.7) in the aspect of security.

Under the temperature 0, the statistics on the non-determinism code generation of algorithm problems and CWE code scenarios in 10 trials are shown in Table 24, Table 25 and Table 26, where Table 26 presents the selected 20 CWE code scenarios. **Cyc.** and **Cog.** represents cyclomatic and cognitive complexities, respectively. Elements in table entries are presented as sets or ratios. - represents an inability to evaluate, including two reasons: tool support is unavailable and the generated code is constant functions, based on the corresponding context (e.g., <*problem 2304*, *Java*> is constant functions and thus cyclomatic and cognitive complexities are not evaluated). From the result, we can observe that when the temperature is set to 0, the statuses of the generated code for each algorithm problem are consistent in 10 trials. The same results are observed in terms of complexity, except for <*problem 2532*, *Python3*>. All generated C code in Problem 304 and 2523 have MissingNullTest vulnerability. We also further manually analyze these code snippets and find that all generated code snippets are completely identical for every <*problem*, *language*> except <*problem 2532*, *Python3*> using different code strictures in different trials. As for the sampled CWE code scenarios, all generated code snippets are also identical for each scenario in 10 trials. Therefore, setting the temperature to 0 may be a potential strategy to mitigate the non-determinism of *ChatGPT* in one-round process.

Result ❷. We also investigate the impact of non-determinism on the multi-round fixing process. For functional correctness and complexity, we sample 20 code snippets with errors randomly from all generated code snippets at temperatures 0.7 and 0, respectively, where each sampled code snippet belongs to one unique <*problem*, *language*>. As

TABLE 22: Functional Correctness, Complexity, and Security for Aft. Problems in 10 Trials at Temperature 0.7

Problem Id	Language	A.	W.A.	C.E.	T.L.E.	R.E.	Cyclomatic				Cognitive				CWE
							L	M	H	V	L	M	H	V	
2124	C	30.0%	70.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%		-			0.0%
	C++	30.0%	70.0%	0.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%		-			0.0%
	Java	0.0%	100.0%	0.0%	0.0%	0.0%	10.0%	90.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%	0.0%
	Python3	20.0%	80.0%	-	0.0%	0.0%	40.0%	60.0%	0.0%	0.0%	20.0%	60.0%	20.0%	0.0%	-
	JavaScript	0.0%	100.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	40.0%	50.0%	10.0%	0.0%	-
2224	C	30.0%	30.0%	40.0%	0.0%	0.0%	0.0%	70.0%	10.0%	20.0%		-			0.0%
	C++	60.0%	0.0%	40.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%		-			0.0%
	Java	80.0%	10.0%	0.0%	10.0%	0.0%	20.0%	70.0%	10.0%	0.0%	20.0%	70.0%	10.0%	0.0%	0.0%
	Python3	70.0%	30.0%	-	0.0%	0.0%	90.0%	10.0%	0.0%	0.0%	20.0%	80.0%	0.0%	0.0%	-
	JavaScript	50.0%	10.0%	-	10.0%	30.0%	30.0%	50.0%	10.0%	10.0%	10.0%	70.0%	20.0%	0.0%	-
2227	C	0.0%	50.0%	30.0%	0.0%	20.0%	0.0%	0.0%	0.0%	100.0%		-			100%
	C++	0.0%	70.0%	20.0%	0.0%	10.0%	75.0%	0.0%	0.0%	25.0%		-			0.0%
	Java	0.0%	80.0%	20.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	70.0%	30.0%	0.0%
	Python3	0.0%	70.0%	-	0.0%	30.0%	0.0%	0.0%	30.0%	70.0%	0.0%	10.0%	70.0%	20.0%	-
	JavaScript	0.0%	100.0%	-	0.0%	0.0%	0.0%	0.0%	80.0%	20.0%	0.0%	30.0%	70.0%	0.0%	-
2264	C	30.0%	30.0%	30.0%	0.0%	10.0%	25.0%	75.0%	0.0%	0.0%		-			42.85%
	C++	0.0%	60.0%	40.0%	0.0%	0.0%	16.7%	33.3%	16.7%	33.3%		-			0.0%
	Java	30.0%	40.0%	30.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	0.0%
	Python3	40.0%	60.0%	-	0.0%	0.0%	40.0%	60.0%	0.0%	0.0%	20.0%	80.0%	0.0%	0.0%	-
	JavaScript	30.0%	70.0%	-	0.0%	0.0%	40.0%	60.0%	0.0%	0.0%	40.0%	60.0%	0.0%	0.0%	-
2304	C	10.0%	50.0%	40.0%	0.0%	0.0%	0.0%	87.5%	12.5%	0.0%		-			0.0%
	C++	50.0%	10.0%	20.0%	0.0%	20.0%	0.0%	100.0%	0.0%	0.0%		-			0.0%
	Java	30.0%	10.0%	60.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	0.0%
	Python3	50.0%	50.0%	-	0.0%	0.0%	70.0%	30.0%	0.0%	0.0%	10.0%	90.0%	0.0%	0.0%	-
	JavaScript	90.0%	10.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	-
2400	C	0.0%	70.0%	0.0%	0.0%	30.0%	70.0%	30.0%	0.0%	0.0%		-			0.0%
	C++	0.0%	10.0%	60.0%	0.0%	30.0%	80.0%	20.0%	0.0%	0.0%		-			0.0%
	Java	0.0%	70.0%	10.0%	0.0%	20.0%	50.0%	50.0%	0.0%	0.0%	40.0%	50.0%	10.0%	0.0%	0.0%
	Python3	0.0%	40.0%	-	0.0%	60.0%	70.0%	30.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	-
	JavaScript	0.0%	100.0%	-	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	60.0%	30.0%	10.0%	0.0%	-
2435	C	50.0%	40.0%	0.0%	0.0%	10.0%	10.0%	80.0%	10.0%	0.0%		-			0.0%
	C++	0.0%	40.0%	60.0%	0.0%	0.0%	0.0%	0.0%	75.0%	25.0%		-			0.0%
	Java	0.0%	90.0%	10.0%	0.0%	0.0%	10.0%	0.0%	80.0%	10.0%	10.0%	0.0%	90.0%	0.0%	0.0%
	Python3	0.0%	90.0%	-	0.0%	10.0%	10.0%	10.0%	70.0%	10.0%	10.0%	70.0%	20.0%	0.0%	-
	JavaScript	0.0%	100.0%	-	0.0%	0.0%	10.0%	10.0%	60.0%	20.0%	10.0%	10.0%	70.0%	10.0%	-
2523	C	10.0%	40.0%	10.0%	40.0%	0.0%	0.0%	0.0%	0.0%	100.0%		-			100%
	C++	20.0%	20.0%	50.0%	0.0%	10.0%	0.0%	20.0%	40.0%	40.0%		-			0.0%
	Java	10.0%	10.0%	60.0%	20.0%	0.0%	60.0%	0.0%	30.0%	10.0%	60.0%	0.0%	30.0%	10.0%	0.0%
	Python3	0.0%	0.0%	-	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	10.0%	90.0%	0.0%	-
	JavaScript	20.0%	40.0%	-	40.0%	0.0%	20.0%	0.0%	40.0%	40.0%	20.0%	0.0%	60.0%	20.0%	-
2532	C	0.0%	10.0%	80.0%	10.0%	0.0%	14.3%	0.0%	0.0%	85.7%		-			0.0%
	C++	0.0%	20.0%	70.0%	0.0%	10.0%	0.0%	33.3%	66.7%	0.0%		-			0.0%
	Java	0.0%	0.0%	90.0%	10.0%	0.0%	90.0%	0.0%	0.0%	10.0%	90.0%	0.0%	10.0%	0.0%	0.0%
	Python3	0.0%	30.0%	-	10.0%	60.0%	50.0%	20.0%	10.0%	20.0%	30.0%	30.0%	30.0%	10.0%	-
	JavaScript	0.0%	70.0%	-	20.0%	10.0%	30.0%	0.0%	10.0%	60.0%	30.0%	0.0%	20.0%	50.0%	-

for security, we select one vulnerable code snippet randomly from each category (selected in this section) of vulnerabilities having generated vulnerable code at temperatures 0.7 and 0, respectively, across algorithm problems and CWE code scenarios. Each error code or vulnerable code is fixed 5 times under the multi-round fixing process. Additionally, the multi-round fixing process, when set to temperatures of 0.7 and 0, is only performed on the generated code snippets in the one-round process at temperatures of 0.7 and 0, respectively. The fixing results are shown in Table 27 - 32. From the results, we can observe the following findings:

▷ **Status Rates.** The data shows that for different trials of the multi-round fixing process at the same error code, the fixing results can be different, regardless of whether the temperature is set at 0.7 or 0. For example, for <*problem 363*, *C*> at temperature 0.7, in the 5 trials, only 2 of them (40%) are successful; for <*problem 2124*, *Python3*>, also only 2 trials are successful.

▷ **Complexity Levels.** The data indicates that the complexity of the fixed code in different trials under the multi-round fixing process with *ChatGPT* may vary, regardless of the temperature setting. For example, the fixed code snippets <*problem 2124*, *Java*> at temperature 0.7 have low and moderate levels in both cyclomatic and cognitive complexities; the fixed code snippets <*problem 2532*, *Python3*> at temperature 0 have complexity levels across low, moderate, high and very high in both cyclomatic and cognitive complexities. In different trials, *ChatGPT* may choose different patches for fixing error code snippets under the multi-round fixing process, even for the setting of temperature 0.

▷ **CWEs.** In different trials under the multi-round fixing process, *ChatGPT* may or may not fix vulnerable code, regardless of the temperature setting. For instance, *ChatGPT* at temperature 0.7 only fixes vulnerable code one time in code scenario 3 of CWE 20; *ChatGPT* at temperature 0 fails to fix vulnerable code one time in code scenario 1 of CWE 190.

Answer to RQ5: Non-determinism of *ChatGPT*

❶ Code generation in one-round process may be affected by *ChatGPT*'s non-determinism factor when the temperature is set to 0.7, resulting in variations of code snippets in functional correctness, complexity, and security. One potential strategy to mitigate the non-determinism of *ChatGPT* in the one-round process is to set the temperature to 0;

❷ However, in the multi-round fixing process, the fixed code snippets by *ChatGPT* may vary in functional correctness, complexity, and security, regardless of the temperature settings of 0.7 and 0.

TABLE 23: Functional Correctness, Complexity, and Security for Bef. Problems in 10 Trials at Temperature 0.7

Problem Id	Language	A.	W.A.	C.E.	T.L.E.	R.E.	Cyclomatic				Cognitive				CWE
							L	M	H	V	L	M	H	V	
9	C	30.0%	0.0%	0.0%	0.0%	70.0%	70.0%	30.0%	0.0%	0.0%		-			0.0%
	C++	40.0%	0.0%	0.0%	0.0%	60.0%	60.0%	40.0%	0.0%	0.0%		-			0.0%
	Java	100.0%	0.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
	Python3	100.0%	0.0%	-	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
	JavaScript	100.0%	0.0%	-	0.0%	0.0%	90.0%	10.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
70	C	100.0%	0.0%	0.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%		-			0.0%
	C++	100.0%	0.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%	0.0%		-			0.0%
	Java	100.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
	Python3	100.0%	0.0%	-	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
	JavaScript	100.0%	0.0%	-	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
304	C	80.0%	0.0%	20.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%		-			100%
	C++	100.0%	0.0%	0.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%		-			0.0%
	Java	100.0%	0.0%	0.0%	0.0%	0.0%	50.0%	40.0%	10.0%	0.0%	80.0%	20.0%	0.0%	0.0%	0.0%
	Python3	100.0%	0.0%	-	0.0%	0.0%	70.0%	30.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
	JavaScript	100.0%	0.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
363	C	10.0%	50.0%	40.0%	0.0%	0.0%	20.0%	20.0%	10.0%	50.0%		-			25%
	C++	30.0%	0.0%	70.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%		-			0.0%
	Java	60.0%	0.0%	40.0%	0.0%	0.0%	40.0%	50.0%	10.0%	0.0%	40.0%	0.0%	60.0%	0.0%	0.0%
	Python3	60.0%	10.0%	-	0.0%	30.0%	20.0%	40.0%	30.0%	10.0%	20.0%	0.0%	50.0%	30.0%	-
	JavaScript	20.0%	40.0%	-	0.0%	40.0%	20.0%	30.0%	40.0%	10.0%	30.0%	0.0%	50.0%	20.0%	-
581	C	100.0%	0.0%	0.0%	0.0%	0.0%	30.0%	50.0%	0.0%	20.0%		-			0.0%
	C++	70.0%	0.0%	30.0%	0.0%	0.0%	28.6%	0.0%	57.1%	14.3%		-			0.0%
	Java	100.0%	0.0%	0.0%	0.0%	0.0%	50.0%	40.0%	0.0%	10.0%	50.0%	40.0%	10.0%	0.0%	0.0%
	Python3	80.0%	10.0%	-	0.0%	10.0%	10.0%	20.0%	50.0%	20.0%	10.0%	50.0%	40.0%	0.0%	-
	JavaScript	100.0%	0.0%	-	0.0%	0.0%	60.0%	30.0%	0.0%	10.0%	60.0%	40.0%	0.0%	0.0%	-
744	C	100.0%	0.0%	0.0%	0.0%	0.0%	80.0%	20.0%	0.0%	0.0%		-			0.0%
	C++	90.0%	0.0%	10.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%		-			0.0%
	Java	100.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
	Python3	100.0%	0.0%	-	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
	JavaScript	100.0%	0.0%	-	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	-
1318	C	100.0%	0.0%	0.0%	0.0%	0.0%	80.0%	20.0%	0.0%	0.0%		-			0.0%
	C++	60.0%	0.0%	40.0%	0.0%	0.0%	50.0%	33.3%	16.7%	0.0%		-			0.0%
	Java	80.0%	0.0%	20.0%	0.0%	0.0%	80.0%	20.0%	0.0%	0.0%	20.0%	60.0%	20.0%	0.0%	0.0%
	Python3	60.0%	40.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	60.0%	30.0%	10.0%	0.0%	-
	JavaScript	100.0%	0.0%	0.0%	0.0%	0.0%	10.0%	90.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	-
1416	C	20.0%	60.0%	10.0%	0.0%	10.0%	0.0%	100.0%	0.0%	0.0%		-			0.0%
	C++	40.0%	30.0%	30.0%	0.0%	0.0%	0.0%	85.7%	14.3%	0.0%		-			0.0%
	Java	40.0%	30.0%	30.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	60.0%	40.0%	0.0%	0.0%	0.0%
	Python3	60.0%	40.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	70.0%	30.0%	0.0%	-
	JavaScript	90.0%	10.0%	-	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	90.0%	10.0%	0.0%	-
2122	C	0.0%	20.0%	60.0%	0.0%	20.0%	22.2%	55.6%	22.2%	0.0%		-			100%
	C++	0.0%	30.0%	50.0%	10.0%	10.0%	0.0%	0.0%	60.0%	40.0%		-			0.0%
	Java	0.0%	60.0%	20.0%	10.0%	10.0%	40.0%	40.0%	10.0%	10.0%	40.0%	40.0%	10.0%	10.0%	0.0%
	Python3	0.0%	50.0%	-	10.0%	40.0%	20.0%	60.0%	20.0%	0.0%	30.0%	40.0%	30.0%	0.0%	-
	JavaScript	0.0%	90.0%	-	10.0%	0.0%	30.0%	40.0%	10.0%	20.0%	40.0%	40.0%	10.0%	10.0%	-

5 Discussion

5.1 Lessons Learnt and Insight

Functionally Correct Code Generation. *ChatGPT* is better at generating functionally correct code for Bef. problems in different languages than Aft. problems. This result indicates that *ChatGPT* may have limitations when generating code for unfamiliar or unseen problems in the training dataset, even if the problems are easy with logic from human perspective. Moreover, *ChatGPT* also has differences in its ability to write code in different languages. In general, the probabilities of *ChatGPT* generating functionally correct code in C++, Java, Python3, and JavaScript are close to each other and substantially higher than that in C.

By analyzing the *ChatGPT*-generated code snippets with errors (i.e., W.A., C.E., R.E., and T.L.E.), we identify several factors for errors in code snippets and unfixed cases under the multi-round fixing process. These findings contribute to the ongoing research focused on improving functionally correct code generation. Among them, besides the need to further strengthen *ChatGPT*'s logical reasoning ability, improving its code generation stability (avoid generating empty body) and alignment with human attention (grasp logical details and meet user requirements such as method signature provided) is also very important, especially for the latter. The code generation process of *ChatGPT* may be careless, and the generated code may fail to meet some of the detailed conditions described, resulting in it being difficult to successfully generate or fix (to functional correct) with the application of the multi-round fixing process. Thus, future research can focus on how to provide additional useful information, such as missing details in the code or the correct algorithm logic, to *ChatGPT* to supplement the multi-round fixing process for fixing, or how to design an effective workflow for automatic code generation.

Code Complexity. The complexity of code generated in different languages may be different. Additionally, the multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets, which may potentially make it increasingly difficult to understand the automatically and consistently generated code by *ChatGPT*.

Secutiry Code Generation. During the evaluation across various scenarios, including algorithm problems and CWE scenarios, it is observed that the code generated by *ChatGPT* often exhibits relevant vulnerabilities, which is a severe issue. However, fortunately, the multi-round fixing process for vulnerable code snippets demonstrates promising results. By providing CWE information, *ChatGPT* is able to automatically fix vulnerable code. Therefore, combining *ChatGPT* with vulnerability detection tools (e.g., *CodeQL*) can mitigate the code generated with vulnerabilities. Furthermore, as an AI-powered assistant learning from large-

TABLE 24: Functional Correctness, Complexity, and Security for Aft. Problems in 10 Trials at Temperature 0

Problem Id	Lg.	Status	Cyc.	Cog.	CWE
	C	W.A.	**M**	-	0.0%
	C++	W.A.	**M**	-	0.0%
2124	Java	W.A.	**M**	**M**	0.0%
	Python3	W.A.	**M**	**M**	-
	JavaScript	W.A.	**M**	**M**	-
	C	A.	**M**	-	0.0%
	C++	A.	**M**	-	0.0%
2224	Java	C.E.	-	-	-
	Python3	A.	**L**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	C.E.	-	-	0.0%
	C++	W.A.	**V**	-	0.0%
2227	Java	W.A.	**V**	**H**	0.0%
	Python3	W.A.	**V**	**H**	-
	JavaScript	W.A.	**H**	**H**	-
	C	W.A.	**L**	-	0.0%
	C++	C.E.	-	-	0.0%
2264	Java	W.A.	**L**	**L**	0.0%
	Python3	W.A.	**M**	**H**	-
	JavaScript	W.A.	**M**	**M**	-
	C	W.A.	**M**	-	0.0%
	C++	C.E.	-	-	0.0%
2304	Java	C.E.	-	-	-
	Python3	A.	**M**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	R.E.	**L**	-	0.0%
	C++	C.E.	-	-	0.0%
2400	Java	W.A.	**M**	**M**	0.0%
	Python3	R.E.	**L**	**L**	-
	JavaScript	W.A.	**M**	**H**	-
	C	A.	**M**	-	0.0%
	C++	C.E.	-	-	0.0%
2435	Java	W.A.	**H**	**M**	0.0%
	Python3	W.A.	**H**	**M**	-
	JavaScript	W.A.	**H**	**H**	-
	C	C.E.	**V**	-	100%
	C++	C.E.	**V**	-	0.0%
2523	Java	C.E.	-	-	0.0%
	Python3	T.L.E.	**H**	**H**	-
	JavaScript	T.L.E.	**V**	**H**	-
	C	C.E.	-	-	0.0%
	C++	C.E.	-	-	0.0%
2532	Java	C.E.	-	-	0.0%
	Python3	W.A.	**V**	**H, V**	-
	JavaScript	W.A.	-	-	-

TABLE 25: Functional Correctness, Complexity, and Security for Bef. Problems in 10 Trials at Temperature 0

Problem Id	Lg.	Status	Cyc.	Cog.	CWE
	C	R.E.	**L**	-	0.0%
	C++	R.E.	**L**	-	0.0%
9	Java	A.	**L**	**L**	0.0%
	Python3	A.	**L**	**L**	-
	JavaScript	A.	**L**	**L**	-
	C	A.	**L**	-	0.0%
	C++	C.E.	-	-	0.0%
70	Java	A.	**L**	**L**	0.0%
	Python3	A.	**L**	**L**	-
	JavaScript	A.	**L**	**L**	-
	C	A.	**M**	-	100%
	C++	A.	**L**	-	0.0%
304	Java	A.	**M**	**M**	0.0%
	Python3	A.	**M**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	C.E.	-	-	0.0%
	C++	C.E.	-	-	0.0%
363	Java	C.E.	-	-	0.0%
	Python3	A.	**H**	**V**	-
	JavaScript	W.A.	**V**	**H**	-
	C	A.	**M**	-	0.0%
	C++	A.	**H**	-	0.0%
581	Java	A.	**M**	**M**	0.0%
	Python3	A.	**H**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	A.	**L**	-	0.0%
	C++	A.	**L**	-	0.0%
744	Java	A.	**L**	**L**	0.0%
	Python3	A.	**L**	**L**	-
	JavaScript	A.	**L**	**L**	-
	C	A.	**L**	-	0.0%
	C++	A.	**L**	-	0.0%
1318	Java	A.	**L**	**M**	0.0%
	Python3	A.	**M**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	W.A.	**M**	-	0.0%
	C++	A.	**M**	-	0.0%
1416	Java	A.	**M**	**H**	0.0%
	Python3	W.A.	**M**	**M**	-
	JavaScript	A.	**M**	**M**	-
	C	C.E.	-	-	0.0%
	C++	C.E.	-	-	0.0%
2122	Java	C.E.	-	-	0.0%
	Python3	W.A.	**M**	**M-**	
	JavaScript	W.A.	-	-	-

TABLE 26: Security Code Generation in 10 Trials at Temperature 0

R.	CWE	Code Scenario	Lg.	Ori.	# Vdn.	# Vln.
1	787	2: *MT.*-5	C	*MT.*	10 (100.0 %)	10 (100.0 %)
3	89	3: [23]-2	Py.	[23]	10 (100.0 %)	0 (0.0 %)
4	20	3: [23]-1	C	[23]	10 (100.0 %)	0 (0.0 %)
5	125	3: [23]-2	C	[23]	10 (100.0 %)	0 (0.0 %)
6	78	1: ExecTainted	C	*CQ.*	10 (100.0 %)	10 (100.0 %)
7	416	3: [23]-1	C	[23]	10 (100.0 %)	0 (0.0 %)
10	434	1: [23]-1	Py.	[23]	10 (100.0 %)	10 (100.0 %)
		2: [23]-2	Py.	[23]	10 (100.0 %)	10 (100.0 %)
		3: [23]-3	Py.	[23]	10 (100.0 %)	0 (0.0 %)
11	476	1: MissingNullTest-a	C	*CQ.*	10 (100.0 %)	10 (100.0 %)
		2: MissingNullTest-b	C	*CQ.*	10 (100.0 %)	10 (100.0 %)
		3: MissingNullTest-c	C	*CQ.*	10 (100.0 %)	10 (100.0 %)
13	190	1: *MT.*-4	C	*MT.*	10 (100.0 %)	10 (100.0 %)
15	798	1: HardcodedCredentials-a	Py.	*CQ.*	10 (100.0 %)	0 (0.0 %)
18	306	1: [23]-1	Py.	[23]	10 (100.0 %)	0 (0.0 %)
		2: [23]-2	Py.	[23]	10 (100.0 %)	0 (0.0 %)
		3: [23]-3	Py.	[23]	10 (100.0 %)	0 (0.0 %)
19	119	2: OverflowBuffer	C	*CQ.*	10 (100.0 %)	0 (0.0 %)
		3: [23]-1	C	[23]	10 (100.0 %)	10 (100.0 %)
30	732	2: DoNotCreateWorldWriteable-b	C	*CQ.*	10 (100.0 %)	0 (0.0 %)

scale datasets, *ChatGPT* itself may also have the ability to detect vulnerabilities, serving as a more flexible vulnerability detection tool.

Non-determinism of *ChatGPT*. By the results in Sec. 4.5, we can observe that code generation may be affected by *ChatGPT*'s non-determinism factor, resulting in variations of code snippets in functional correctness, complexity, and security. One potential strategy to mitigate this issue is to set *ChatGPT*'s temperature to 0. However, this strategy can only work in the one-round process. In the multi-round fixing process, the fixed code snippets by ChatGPT may vary in functional correctness, complexity, and security, regardless of the temperature settings of 0.7 and 0. There are also other strategies to mitigate the non-determinism factor and even find better results by adjusting various hyperparameters in LLMs such as prompts. However, in this study, we do not adjust them and it is out of the paper's scope since we focus on simulating real-world usage scenarios of *ChatGPT* in the code generation task. We plan to investigate LLM-based code generation in various aspects across different settings to hyperparameters in our future work. Such an investigation would allow us to better understand the non-determinism of LLMs in code generation with different hyperparameter settings and may potentially help improve the quality of the code snippets generated by LLMs.

Impact Analysis of Token Limitation. The token limitation fo *ChatGPT* may influence the output from *ChatGPT*. If the total length of the input prompt and output content from *ChatGPT* exceeds the limitation, the excess part is discarded, which may result in an incomplete code snippet. To avoid this problem, we propose a *token-limitation* strategy

TABLE 27: Multi-round Fixing Process for Algorithm Problems in 5 Trials at Temperature 0.7

Problem Id	Lg.	A. Rate	Cyc.	Cog.
9	C	100%	**L**	-
	C++	100%	**L, M**	-
363	C	40%	**H, V**	-
	C++	100%	**M, H**	-
	Java	100%	**M, H**	**H**
	JavaScript	60%	**H, V**	**V**
744	C++	100%	**L, M**	-
1416	C	0.0%	**M, H**	-
	C++	0.0%	**M, H, V**	-
	Java	100%	**M**	**M**
2122	C	0.0%	**H, V**	-
	C++	0.0%	**M, H, V**	-
2124	Java	100%	**L, M**	**L, M**
2224	C++	60%	**L, M, V**	-
	Python3	0.0%	**L, M**	**L, M, H**
2227	Java	0.0%	**V**	**H, V**
2400	C++	0.0%	**L, M, H**	-
2523	Python3	60%	**H, V**	**H, V**
	JavaScript	80%	**H, V**	**H**
2532	Java	0.0%	**M, H, V**	**L, H**

TABLE 28: Multi-round Fixing Process for Algorithm Problems in 5 Trials at Temperature 0

Problem Id	Lg.	A. Rate	Cyc.	Cog.
9	C	100%	**L**	-
	C++	100%	**L**	-
363	C++	100%	**M**	-
	Java	100%	**M**	**H**
1416	Python3	0.0%	**M**	**M**
2122	C	0.0%	**V**	-
	Python3	0.0%	**M**	**M**
2124	C++	20%	**M**	-
	Java	100%	**L**	**M**
	Python3	40%	**L, M**	**L, M**
2227	C++	0.0%	**V**	-
2264	C	100%	**M**	-
	Python3	0.0%	**M**	**M**
	JavaScript	100%	**M**	**L, M**
2435	C++	0.0%	**H, V**	-
	Python3	0.0%	**M, H**	**L, M, H**
	JavaScript	0.0%	**H**	**H**
2532	C	0.0%	**M, V**	-
	Java	0.0%	**M, H**	**M, H**
	Python3	0.0%	**L, M, H, V**	**L, M, H, V**

TABLE 29: Multi-round Fixing Process for CWE of Algorithm Problems in 5 Trials at Temperature 0.7

Problem Id	Lg.	# Fixed.
304	C	5/5 (100.0 %)
363	C	5/5 (100.0 %)
2122	C	5/5 (100.0 %)
2227	C	5/5 (100.0 %)
2264	C	5/5 (100.0 %)
2523	C	5/5 (100.0 %)

TABLE 30: Multi-round Fixing Process for CWE of Algorithm Problems in 5 Trials at Temperature 0

Problem Id	Lg.	# Fixed.
304	C	5/5 (100.0 %)
2523	C	5/5 (100.0 %)

TABLE 31: Multi-round Fixing Process for Security Code Generation in 5 Trials at Temperature 0.7

R.	CWE	Code Scenario	Lg.	Ori.	# Fixed.
1	787	2: *MT.*-5	C	*MT.*	5/5 (100.0 %)
3	89	3: [23]-2	Py.	[23]	5/5 (100.0 %)
4	20	3: [23]-1	C	[23]	1/5 (20.0 %)
6	78	1: ExecTainted	C	*CQ.*	5/5 (100.0 %)
10	434	1: [23]-1	Py.	[23]	5/5 (100.0 %)
		2: [23]-2	Py.	[23]	3/5 (60.0 %)
11	476	1: MissingNullTest-a	C	*CQ.*	5/5 (100.0 %)
		2: MissingNullTest-b	C	*CQ.*	5/5 (100.0 %)
		3: MissingNullTest-c	C	*CQ.*	5/5 (100.0 %)
13	190	1: *MT.*-4	C	*MT.*	5/5 (100.0 %)
15	798	1: HardcodedCredentials-a	Py.	*CQ.*	4/5 (80.0 %)
18	306	1: [23]-1	Py.	[23]	5/5 (100.0 %)
19	119	3: [23]-1	C	[23]	5/5 (100.0 %)

TABLE 32: Multi-round Fixing Process for Security Code Generation in 5 Trials at Temperature 0

R.	CWE	Code Scenario	Lg.	Ori.	# Fixed.
1	787	2: *MT.*-5	C	*MT.*	5/5 (100.0 %)
6	78	1: ExecTainted	C	*CQ.*	5/5 (100.0 %)
10	434	1: [23]-1	Py.	[23]	5/5 (100.0 %)
		2: [23]-2	Py.	[23]	0/5 (0.0 %)
11	476	1: MissingNullTest-a	C	*CQ.*	5/5 (100.0 %)
		2: MissingNullTest-b	C	*CQ.*	0/5 (0.0 %)
		3: MissingNullTest-c	C	*CQ.*	5/5 (100.0 %)
13	190	1: *MT.*-4	C	*MT.*	4/5 (80.0 %)
19	119	3: [23]-1	C	[23]	5/5 (100.0 %)

to discard as little of the earliest content and supplement necessary information throughout the conversation. This strategy avoids missing the necessary details in the code generation task for *ChatGPT*. To further analyze the impact of token limitation on code generation. We leverage the selected algorithm problems and CWE code scenarios in Sec. 4.5 and set *ChatGPT*'s temperature to 0 to stabilize the output in the one-round process. We first count the token used for output in each code generation to these selected problems and scenarios and then limit the maximum output token to half of the counted token usage in each corresponding code generation for generating incomplete code snippets. For example, if *ChatGPT* outputs x tokens to the problem/scenario i, then limit the maximum output token to $x/2$ to i for generating code snippet. This approach simulates the case of generating incomplete code snippets due to exceeding the token limitation of *ChatGPT* and provides the pairs of the complete code snippets and the corresponding incomplete ones. After testing and manually analyzing these generated incomplete code snippets, we find that all these code snippets have compile or syntax errors due to the discarded part (e.g., a binary operator has no right operand). Additionally, the cyclomatic and cognitive complexities of the incomplete code snippets are also less than or equal to that of the original code snippets (e.g., an incomplete code snippet misses one `if` statement to the corresponding original code snippet or the discarded part does not contain any branch statement). For security, some of the code's vulnerabilities are no longer present (the vulnerable parts are in the discarded part), but some code still retains its vulnerabilities (the vulnerable parts are not discarded).

Comparison with Other Code Generation Models. In the realm of code generation, recent advancements have been significantly driven by LLMs trained on extensive code datasets. Code-related LLMs, such as *Codex* [9] and *CodeGen* [67], have demonstrated substantial capabilities and generalization in code generation tasks. These models generate code by autoregressively predicting the next token from the previous context (e.g., function signature, docstring, and previously generated tokens) and combining the previous context and the generated tokens together as the finally generated code snippet. *ChatGPT* takes this a step further. It has also demonstrated impressive code generation capabilities. Additionally, with RLHF [31], ChatGPT supports the ability to answer follow-up questions, providing even more powerful and versatile features compared to previous code-related LLMs.

5.2 Limitations

The results and experiments of this study are limited to two parts: (1) *ChatGPT* is a closed-source model, which means that we are unable to directly map the analysis results to the internal workings of *ChatGPT* or understand the specific characteristics of the model. Furthermore, the exact training data used by ChatGPT remains unknown to us. Consequently, it becomes difficult to ascertain whether the problems we input have been previously used in the training dataset; (2) it is important to note that *ChatGPT* is a continuously evolving and training model. The responses generated by *ChatGPT* in this study can only reflect the performance of the model at the time of our work (i.e., model version *gpt-3.5-turbo-0301* of *ChatGPT*).

5.3 Threats to Validity

***LeetCode* Problems and CWE Scenarios.** To reduce bias by manually selecting subjects for evaluation, we utilize *LeetCode* problems as our main dataset. However, *LeetCode* problems are designed specifically for coding practice and interview preparation. While they cover a range of programming concepts and challenges, they may not fully represent the complexity and diversity of real-world coding tasks. Real-world coding scenarios often involve various external factors, domain-specific requirements, and specific constraints that may not be fully captured by *LeetCode* problems alone. Moreover, the classes of vulnerabilities that *LeetCode* problems' code can have are limited, so we also utilize CWE scenarios [23] to supplement the evaluation of ChatGPT's security code generation. Nevertheless, similar to *LeetCode* problems, these scenarios may not cover all real-world code scenarios.

***LeetCode* Online Judgment.** *LeetCode* online judgment platform terminates the testing process upon encountering the first failed test case. Thus, the test case pass rates provided by the platform may serve as a lower bound, but it does not affect the statuses of code snippets generated by *ChatGPT* and the conclusion drawn from this study.

Vulnerability Detection by *CodeQL*. *CodeQL* may report a code as vulnerable when it is actually secure. To mitigate the risk, human expertise is employed to manually inspect the code for potential vulnerabilities, thereby ensuring the accuracy and reliability of the analysis.

Limited Languages in Vulnerability Detection. The evaluation of vulnerability detection in our study only focuses on limited languages (C, C++, and Java to *LeetCode* problem scenarios and C and Python to CWE scenarios) out of five languages (C, C++, Java, Python, and JavaScript) due to the targeted scenarios and limitations of vulnerability detection tools. Though the evaluation results provide insights into *ChatGPT*-based code generation in these languages, our study does not fully reflect the spectrum of all five languages in security.

Statistical Validity. *ChatGPT* has randomness. When faced with the same input prompt, *ChatGPT* may produce different responses. To reduce the risk, we use 728 *LeetCode* problems. For each <*Problem, Language*> pair, we independently generate one corresponding code snippet once[26], following the law of large numbers. For CWE's code scenarios, we generate 60 code snippets independently for each scenario. As for the multi-round process, we sample many code snippets for experimentation and we set 5 to the maximum round number [53], a reasonable round limit.

6 Related Work

Language Models. Language models have a wide range of applications in NLP, including machine translation, question answering, summarization, text generation, code generation and so on [16], [18], [68], [69], [70], [71], [72], [73], [9], [4]. These models, with a large number of parameters, are trained on extensive corpus to better understand language (i.e., LLM). One of the fundamental architectures used in language models is Transformer [6], which consists of stacked encoders and decoders. Transformer utilizes self-attention mechanism to weigh the importance of words in the input text, capturing long-range dependencies and relationships between words. Many language models are built upon Transformer. *ELMo* [74] employs multi-layer bidirectional LSTM (long short-term memory) and provides high-quality word representations. *GPT* [32] and *BERT* [29] are based on the decoder (unidirectional) and encoder (bidirectional) components of the Transformer, respectively. They utilize pre-training and fine-tuning techniques. *GPT-2* [33] and *GPT-3* [20] are the successors of *GPT*, with *GPT-2* having a larger model size in parameters than *GPT*, and *GPT-3* being even larger than *GPT-2*. Additionally, with larger corpus, *GPT-2* and *GPT-3* introduce zero-shot and few-shot learning to enable adaptation to multitask scenarios. *Codex* [9] is obtained by training *GPT-3* on GitHub code data. It serves as the underlying model for GitHub *Copilot* [11], a tool that can automatically generate and complete code automatically. *InstructGPT* [31] uses additional supervised learning and reinforcement learning from human feedback (RLHF) to fine-tune *GPT-3*, aligning the language model with users. *ChatGPT* [12], based on *GPT-3.5* [38], utilizes the same methods as *InstructGPT* and provides the ability to answer follow-up questions.

Code Generation. Code generation [75] is a fundamental application of language models that aims to automatically generate or complete computer code based on given specifications or natural language descriptions, improving programming productivity. There is a lot of research on it, including traditional approaches and AI-based approaches.

26. *ChatGPT* has a rate-limiting of queries and it may be retrained at a later date. Thus, we query once for each problem and do not requery for failed responses (e.g., response is empty or irrelevant such as violation of policy. This small amount of responses is excluded from the experimental evaluation).

Traditional code generation [75], [14], [76], [77], [78], [79] approaches typically rely on predefined templates or rules (e.g., context-free grammar), along with input-output specifications, which limits their flexibility and requires manual effort. For example, Gulwani [78] identifies a string expression language available to various string manipulation tasks (e.g., extract substrings in a specific format) and designs an algorithm for learning a string expression that is consistent with the provided input-output examples. As for AI-based approaches [15], [9], [11], [80], [12], [81], [82], [67], they leverage deep learning and NLP to overcome these limitations and can offer more intelligent and adaptable code-generation capabilities. Li et al. [80] leverage recurrent neural network with attention mechanism and pointer mixture network on abstract syntax tree (AST) to predict next word in code completion tasks, learning from large-scale codebases. Liu et al. [83] model the structural information in AST and use Transformer-XL network and multi-task learning to capture long-term dependency in programs and learn two disjoint code-related tasks in code completion, respectively. Ashwin et al. [84] combine traditional method with neural network (e.g., LSTM network) to generate code from examples, with high correctness, strong generalization, and low synthesis time. Recently, with the advantages of LLMs, researchers apply LLMs directly to the code generation task by using extensive code datasets, such as *Codex* [9], *Copilot* [11], *CodeGen* [67], providing more powerful capabilities and generalizations. These code-related LLMs (e.g., *Codex*) autoregressively predict the next token from the previous context (e.g., function signature, docstring, and previously generated tokens) in code generation and combine the previous context and the generated tokens together as the finally generated code snippet. *ChatGPT* [12], the state-of-the-art LLM based on *GPT-3.5* [21], has also demonstrated impressive code generation capabilities. Additionally, with RLHF [31], *ChatGPT* supports the ability to answer follow-up questions, providing even more powerful and versatile features compared to previous code-related LLMs.

Evaluation on LLM-based Code Generation. Hendrycks et al. [85] craft APPS benchmark of Python programming problems and assess the code generation performance for several *GPT*-based variant models by fine-tuning with APPS. Fan et al. [51] systematically study whether automated program repair (APR) techniques, including *Codex*, can fix the incorrect solutions to *LeetCode* problems produced by *Codex*. Xia et al. [86] perform an extensive study on directly applying LLMs (9 state-of-the-art LLMs) for APR. They evaluate different ways of using LLMs for the task, including the entire-patch fix, the chunk-of-code fix, and the single-line fix. Pearce et al. [66] examine the use of LLMs (e.g., *Codex*) by zero-shot learning for vulnerability repair. *CodeT* [87] utilizes LLMs to generate functionally correct code solutions by leveraging dual execution agreement. It generates multiple code solutions and multiple test cases for a given programming problem and executes the generated code solutions using the generated test cases to rank and find the best solution. Dong et al. [53] introduce the concept of software development life cycle and propose a self-collaboration framework that leverages different *ChatGPT* conversations to play different roles (e.g., analyst, developer, and tester), collaborating to generate code. Sobania et al. [40] conduct an evaluation of *Copilot* on standard program synthesis benchmark program, comparing the results with genetic programming. Pearce et al. [23] assess *Copilot*'s security code generation on the top-25 CWE vulnerabilities. Nguyen et al. [24] evaluates the quality of *Copilot*-generated code by using 33 *LeetCode* problems in 4 different languages. Kou et al. [88] investigate the attention alignment between the nature language description from humans and the code generation by LLMs. Liu et al. [25] propose *EvalPlus* framework to enhance code generation benchmarks. *EvalPlus* takes in a base evaluation dataset and uses LLMs and mutation technique to produce and diversify large amounts of new test cases. Liu et al. [89] characterize several code quality issues of *ChatGPT*-based code generation across Java and Python languages, including correctness and maintainability. They also examine the ability of *ChatGPT* to repair bugs and code style issues by leveraging feedback information. Different from their work, we conduct a systematical assessment with deep analysis for *ChatGPT*-based code generation across five languages in terms of correctness, complexity, and security, including the multi-round process. Our research significantly extends the current understanding of *ChatGPT*-based code generation. We not only evaluate the correctness of the generated code but also provide a deep dive into the underlying causes of incorrectness in *ChatGPT*-generated code. We also deeply assess the complexity and security of the generated code. Furthermore, we deeply investigate the impact of the multi-round fixing process on these aspects, providing a more realistic evaluation of *ChatGPT*'s capabilities in iterative code generation scenarios. This comprehensive study underscores the practical implications of AI-generated code in real-world software development.

7 Conclusion

In this paper, we present a systematic assessment of *ChatGPT*-based code generation. We comprehensively evaluate code snippets generated by *ChatGPT* from three aspects of correctness, complexity, and security, including the multi-round fixing process. Our experimental results demonstrate that (1) *ChatGPT* is better at generating functionally correct code for Bef. problems in different languages than Aft. problems (the average *Accepted* rate of the former exceeds the latter by 48.14%), but *ChatGPT*'s ability to directly fix erroneous code to achieve correct functionality is relatively weak; (2) the distribution of cyclomatic and cognitive complexity levels for code snippets in different languages varies. Additionally, the multi-round fixing process with *ChatGPT* generally preserves or increases the complexity levels of code snippets; (3) in algorithm scenarios with languages of C, C++, and Jave, and CWE scenarios with languages of C and Python3, the code generated by *ChatGPT* has relevant vulnerabilities. Fortunately, the multi-round fixing process for vulnerable code snippets demonstrates promising results, with a high percentage (100% and 89.4%) of vulnerabilities successfully addressed; and (4) code generation may be affected by *ChatGPT*'s non-determinism factor, resulting in variations of code snippets in functional correctness, complexity, and security. Overall, our findings uncover potential issues and limitations that arise in the *ChatGPT*-based code generation and pave the way for improving AI and LLM-based code generation techniques.

References

[1] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017.

[2] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proceedings of The Web Conference 2020*, 2020, pp. 2309–2319.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[4] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[5] M. Tufano, D. Drain, A. Svyatkovskiy, S. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context." *arXiv: Software Engineering,arXiv: Software Engineering*, Sep 2020.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[7] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[8] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[11] Github, "Github copilot · your ai pair programmer," 2023, https://github.com/features/copilot.

[12] OpenAI, "Chatgpt: Optimizing language models for dialogue," 2023, https://openai.com/blog/chatgpt/.

[13] Github, "What is ai code generation?" 2023, https://resources.github.com/artificial-intelligence/what-is-ai-code-generation/.

[14] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[15] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[16] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models." in *USENIX Security Symposium*, vol. 6, 2021.

[17] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," 2007.

[18] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, 2022.

[19] R. Nagata, M. Kimura, and K. Hanawa, "Exploring the capacity of a large-scale masked language model to recognize grammatical errors," *arXiv preprint arXiv:2108.12216*, 2021.

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[21] OpenAI, "Gpt-3.5 models," 2023, https://platform.openai.com/docs/models/overview.

[22] ——, "Gpt-4 technical report," 2023.

[23] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[24] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[25] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[26] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, and J. Yu, "Security weaknesses of copilot generated code in github," *arXiv preprint arXiv:2310.02059*, 2023.

[27] LeetCode, "The world's leading online programming learning platform," 2023, https://leetcode.com/.

[28] "Online artifact," 2024, https://zenodo.org/records/10556350.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[31] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[32] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[33] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[34] mitre, "2022 cwe top 25 most dangerous software weaknesses," 2022, https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html #cwe_top_25.

[35] GitHub, "Codeql documentation," 2023, https://codeql.github.com/docs/.

[36] OpenAI, "openai-cookbook," 2023, https://github.com/openai/openai-cookbook.

[37] "Dall·e, gpt, midjourney, stable diffusion, chatgpt prompt marketplace," 2023, https://promptbase.com/.

[38] "Openai," https://chat.openai.com/, 2023, accessed: 2023-08-30.

[39] "Reverse engineered chatgpt api by openai. extensible for chatbots etc." https://github.com/acheong08/ChatGPT, 2023, accessed: 2023-04-30.

[40] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming," in *Proceedings of the genetic and evolutionary computation conference*, 2022, pp. 1019–1027.

[41] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[42] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 71–82.

[43] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[44] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[45] M. P. Fay and M. A. Proschan, "Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules," *Statistics surveys*, vol. 4, p. 1, 2010.

[46] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.

[47] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[48] kamyu104, "Python / modern c++ solutions of all 2577 leetcode problems," 2023, https://github.com/kamyu104/LeetCode-Solutions.

[49] Y. Hu, Y. Fang, Y. Sun, Y. Jia, Y. Wu, D. Zou, and H. Jin, "Code2img: Tree-based image transformation for scalable code clone detection," *IEEE Transactions on Software Engineering*, 2023.

[50] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[51] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," *arXiv preprint arXiv:2205.10583*, 2022.

[52] Codeforces, "Programming competitions and contests, programming community," 2023, https://codeforces.com/.

[53] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.

[54] M. Rigaki and S. Garcia, "A survey of privacy attacks in machine learning," *arXiv preprint arXiv:2007.07646*, 2020.

[55] H. Hu, Z. Salcic, L. Sun, G. Dobbie, P. S. Yu, and X. Zhang, "Membership inference attacks on machine learning: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–37, 2022.

[56] C. E. C. Dantas and M. A. Maia, "Readability and understandability scores for snippet assessment: an exploratory study," *arXiv preprint arXiv:2108.09181*, 2021.

[57] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 417–427.

[58] SonarSource, "Clean code for teams and enterprises with sonarqube," 2023, https://www.sonarsource.com/products/sonarqube/.

[59] sarnold, "Cccc project documentation," 2023, https://sarnold.github.io/cccc/.

[60] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[61] sonarsource, "cognitive complexity," 2023, https://www.sonarsource.com/resources/cognitive-complexity/.

[62] pmd, "An extensible cross-language static code analyzer," 2023, https://pmd.github.io/.

[63] GitHub, "Codeql full cwe coverage," 2023, https://codeql.github.com/codeql-query-help/full-cwe/.

[64] W. Stallings, L. Brown, M. D. Bauer, and M. Howard, *Computer security: principles and practice*. Pearson Upper Saddle River, 2012, vol. 3.

[65] K. Krishna, Y. Chang, J. Wieting, and M. Iyyer, "Rankgen: Improving text generation with large ranking models," *arXiv preprint arXiv:2205.09726*, 2022.

[66] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1–18.

[67] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[68] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[69] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan, "DIALOGPT : Large-scale generative pre-training for conversational response generation," in *Proc. of ACL*, 2020.

[70] J. Pilault, R. Li, S. Subramanian, and C. Pal, "On extractive and abstractive neural document summarization with transformer language models," in *Proc. of EMNLP*, 2020, pp. 9308–9319.

[71] X. Cai, S. Liu, J. Han, L. Yang, Z. Liu, and T. Liu, "Chestxraybert: A pretrained language model for chest radiology report summarization," *IEEE Transactions on Multimedia*, pp. 845 – 855, 2021.

[72] D. Khashabi, S. Min, T. Khot, A. Sabharwal, O. Tafjord, P. Clark, and H. Hajishirzi, "Unifiedqa: Crossing format boundaries with a single qa system," *arXiv preprint arXiv:2005.00700*, 2020.

[73] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[74] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations. arxiv 2018," *arXiv preprint arXiv:1802.05365*, vol. 12, 2018.

[75] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[76] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *International conference on machine learning*. PMLR, 2016, pp. 2933–2942.

[77] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 27–38.

[78] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[79] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Counterexample guided inductive synthesis modulo theories," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 270–288.

[80] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[81] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[82] "Try bard, an ai experiment by google," 2023, https://bard.google.com/.

[83] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 37–47.

[84] K. Ashwin, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani, "Neural-guided deductive search for real-time program synthesis from examples," *International Conference on Learning Representations,International Conference on Learning Representations*, Feb 2018.

[85] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[86] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[87] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[88] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Is model attention aligned with human attention? an empirical study on large language models for code generation," *arXiv preprint arXiv:2306.01220*, 2023.

[89] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *arXiv preprint arXiv:2307.12596*, 2023.