

# Unseen Horizons: Unveiling the Real Capability of LLM Code Generation Beyond the Familiar

Yuanliang Zhang\*, Yifan Xie\*, Shanshan Li†, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, Xiangke Liao

College of Computer Science and Technology  
National University of Defense Technology  
Changsha, China

{zhangyuanliang13, xieyifan, shanshanli, liuke23, jiazhouyang, xbhuang, songj19, luochaopeng18, zhengzhizheng23, xurulin11, liuyitong22, xkliao}@nudt.edu.cn, {ridicious1997, si.zheng1009}@gmail.com

**Abstract**—Recently, large language models (LLMs) have shown strong potential in code generation tasks. However, there are still gaps before they can be fully applied in actual software development processes. Accurately assessing the code generation capabilities of large language models has become an important basis for evaluating and improving the models. Some existing works have constructed datasets to evaluate the capabilities of these models. However, the current evaluation process may encounter the illusion of “Specialist in Familiarity”, primarily due to three gaps: the exposure of target code, case timeliness, and dependency availability. The fundamental reason for these gaps is that the code in current datasets may have been extensively exposed and exercised during the training phase, and due to the continuous training and development of LLM, their timeliness has been severely compromised.

The key to solve the problem is to, as much as possible, evaluate the LLMs using code that they have not encountered before. Thus, the fundamental idea in this paper is to draw on the concept of code obfuscation, changing code at different levels while ensuring the functionality and output. To this end, we build a code-obfuscation based benchmark OBFUSEVAL. We first collect 1,354 raw cases from five real-world projects, including function description and code. Then we use three-level strategy (symbol, structure and semantic) to obfuscate descriptions, code and context dependencies. We evaluate four LLMs on OBFUSEVAL and compared the effectiveness of different obfuscation strategy. We use official test suites of these projects to evaluate the generated code. The results show that after obfuscation, the average decrease ratio of test pass rate can up to 62.5%.

**Index Terms**—Large Language Model, Code Generation Capability, Code Dataset

## I. INTRODUCTION

With the rapid development of the Large Language Model (LLM), the code generation capability of LLMs has attracted lots of attention [1]–[8]. However, how to accurately assess the code generation capability of LLMs in production-level software development is still an open question. A variety of benchmark tests for code generation have been proposed, however, there are still gaps between these benchmark tests and the actual software development process.

Traditional datasets as evaluation benchmarks [9]–[12] play key roles in evaluating the capabilities of LLMs. However,

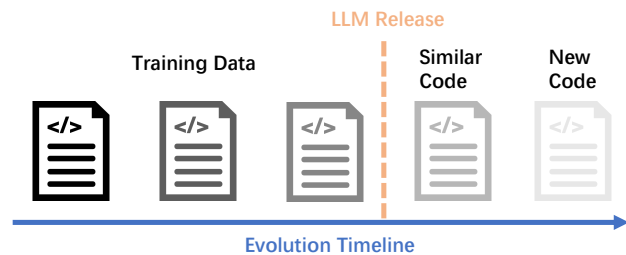


Fig. 1: The familiarity level of code to LLM. The more transparent elements indicate less familiarity.

they focus mainly on standalone functions based on algorithmic problems, which cannot reflect the complexity of real software development, as LLMs may face potential challenges in handling code that is interdependent with other contextual elements of the project [13]. There have been several benchmarks [13]–[16] which are built based on real-world production projects, while there remain three gaps for these benchmarks to objectively evaluate the code generation capability of LLM: exposure of target code, case timeliness, and dependency availability.

**Gap 1: Target code has been exposed in the pre-training stage.** Previous benchmarks [13], [15], [16] have only rewritten the functional descriptions without modifying the code itself. This could result in the target code being exposed to LLMs during training, making the evaluation results not objective as the code has been exercised extensively.

**Gap 2: Collected case is time-sensitive.** With the rapid development of LLMs, data will be continuously trained. Although previous benchmark [14] has collected modified code from history after LLMs training, the modified code may still have been exposed to LLMs because of the existence of code clones. In addition, benchmarks that rely on historical data will suffer from the timeliness problem and cannot be used to evaluate subsequent releases of LLMs.

**Gap 3: Precise dependencies are difficult to provide in real usage scenarios.** Existing benchmarks [13], [15] directly provide the model with all the dependencies needed to generate the target code. However, such conditions can not be always satisfied in real development.

\*Co-first authors

†Corresponding author

**In conclusion, existing evaluation process may suffer from the “Specialist in Familiarity” problem**, which means LLMs can perform well on code that they are very familiar with, often due to extensive experience or repeated exposure. This term highlights the individual’s strength in specific, well-known domains, but it also implies that their expertise may not extend beyond these familiar areas. The primary reason is that the code used to evaluate LLMs may be extensively exposed and exercised for training. As shown in Fig. 1, the unfamiliarity of code to LLM is increased along with the timeline. Unfortunately, even for the code collected after LLM training and release may be similar to the training data due to the existence of code reuse.

To solve these problems, we need to evaluate using code that LLMs have not encountered before as much as possible. To this end, we propose a new obfuscation-based benchmark OBFUSEVAL to evaluate LLMs’ code generation capability. Our main target is to evaluate the LLM on the code generation tasks that it has never encountered before and simulate real software development process. OBFUSEVAL has three key characteristics: 1) **To solve Gap 1**, we collect highly starred projects from GitHub [17] and select functions from them that are introduced after a certain time point. These functions are covered by the official test suites. 2) **To solve Gap 2**, all the raw data have been obfuscated by different-level strategies (symbol, structure, and semantic) to rewrite both functional descriptions and code. The process of code obfuscation can ensure that datasets are reused without the concern that they might still become training data in the future (the code obfuscation process can be repeated). 3) **To solve Gap 3**, we provide relevant code dependencies in a compromise manner, simulating real-world development scenarios without deliberately sacrificing the generation capabilities of LLM. We identify all the contextual dependencies necessary for each function, including necessary API calls, structures, macros, etc., and also add some dependencies that are not related to the target function code for obfuscation.

To effectively utilize our dataset and evaluate LLM’s real capability on generating unfamiliar code, we built a project-level execution platform that provides an off-the-shelf runtime environment to automatically evaluate the functional correctness of the generated code. We developed this platform based on Docker, cloning and building the environment for all projects. Given a model-generated code, the code will automatically replace the original code. Then the projects will be compiled and tested to see whether there are compilation errors or test failures.

We comprehensively evaluated four state-of-the-art code generation models (ChatGPT3.5, ChatGPT4-1106, ChatGPT4-0125, and DeepSeek-Coder-V2) on OBFUSEVAL. We analyzed each model’s effectiveness under different obfuscation strategies. The results show that after code obfuscation, the average decrease ratio of test pass rate can up to 62.5%. In addition, we found that even passing all the tests, code generated by LLMs may still suffer from non-functional code issues (e.g., code robustness), which can guide the developers to better discern

and utilize the code generated by LLMs.

The main contributions of the paper are as follows:

- We reveal that existing benchmarks are insufficient for objectively evaluating the code generation capabilities of LLMs, primarily due to three gaps: exposure of target code, case timeliness, and dependency availability.
- We propose an obfuscation-based approach to rewrite the functional descriptions, code, and dependencies to prevent the target code from being exposed in the training stage. We design different levels of obfuscation strategies and examine their effectiveness. Future research can design sophisticated obfuscation process to better explore the potential of LLM’s capability based on our results.
- We build an obfuscation-based benchmark OBFUSEVAL<sup>1</sup> using code from real-world projects. We evaluated four state-of-the-art code generation models on OBFUSEVAL. The results show that after code obfuscation, the average decrease ratio of test pass rate is 15.3%-62.5%, demonstrating the inflated capabilities of LLMs. We also identify non-functional code issues in the passed cases which can be studied in future work.

## II. BACKGROUND

In this section, we first conduct a comprehensive examination of the latest advancements in Large Language Models (LLMs) within code generation. Subsequently, we delve into the related work of evaluations crafted for code generation, along with the limitations and challenges encountered in assessing the performance of LLMs. Finally, we introduce the motivation for incorporating code obfuscation in the evaluation of large language models.

### A. Large Language Models for Code Generation.

The process of code generation, which involves the automatic creation of complete program code or the completion of code snippets from higher-level representations, such as natural language descriptions, models, or specifications, plays a pivotal role in enhancing programming efficiency and mitigating human error [18]–[21]. Recent advancements in Large Language Models (LLMs) for code generation have garnered significant attention in the realm of computer science research. These LLMs, such as GPT-4 [22], ChatGLM [23], CODEX [9], and CodeGen [24], have demonstrated remarkable capabilities not only in general natural language processing tasks [25] but also in the specific area of code generation. Notably, GPT-4 achieved the highest pass rate on the HumanEval benchmark [9], indicating a growing trend to evaluate the code generation capacity of general LLMs [26].

Code-specific LLMs, which are trained primarily on massive code-specific corpora, often outperform general LLMs in code generation tasks [26]–[33]. Diverse training approaches have been employed, with some models like InCoder [34] and StarCoder [35] being trained with the “filling-in-the-middle” capability for infilling missing code based on context. Varieties

<sup>1</sup> <https://github.com/zhangbuzhang/ObfusEval>

TABLE I: LLM’s code generation performance on code competition problems of different time

OJ Website	GPT3.5-turbo (2023.06)		GPT4.0-preview (2023.11)	
	zero-shot	few-shot	zero-shot	few-shot
LeetCode 2023.11-2024.01	19.7%	22.0%	39.4%	40.9%
LeetCode 2018.09-2018.11	95.6%	93.3%	96.7%	95.6%

of code LLMs have been proposed, such as WizardCoder [29], Instruct-StarCoder [36], and Instruct-CodeGen [37], each designed with different training objectives.

### B. Evaluations for LLM’s Code Generation

**Benchmark construction.** Current benchmarks built based on real projects usually rewrite only the functional descriptions without modifying the code [11], [13]–[15], [38], [39], which may lead to the “code leak” issue. SWE-BENCH [14] collects modified code from the project’s history. However, the code may be in the training set of subsequent releases of LLMs. Future LLMs may become experts in solving problems in SWE-BENCH, but they may still struggle with new code problems in real-world scenarios. EvoCodeBench [40] periodically update the dataset, but there is still possibility of introducing similar code due to the existence of code clones.

**Context dependencies.** Traditional benchmarks focus on generating independent code units, ignoring the contextual relationships between code [9], [41]–[45]. For example, SWE-BENCH does not provide the complete dependencies of the generated code, so it is hard to distinguish the capability of LLMs to generate the target code and its dependencies. However, current studies indicate that only about 30% of methods in open-source projects are relatively independent [13] in real-world scenarios, methods often depend on each other or share variables, which is not considered in these traditional benchmarks. Some works [13], [15] provide complete dependencies but do not include useless and obfuscated dependencies. This discrepancy does not align with software development scenarios and fails to accurately measure the ability of LLMs to assist developers in practical settings. Therefore, we need evaluation methods closer to real-world scenarios to comprehensively assess LLMs’ performance in real-world software development.

**Evaluation methods.** When evaluating the code generation capabilities of LLMs, existing studies and benchmarks focus on the basic correctness of the code, which is usually verified by executing simple test cases (e.g., unit tests) [46]–[48]. In our work, we try to assess both syntactic and functional correctness of LLMs-generated code in real scenarios, by leveraging both compile checking and systematic testing, which aligns more closely with the requirements of real-world development. In addition, previous benchmarks were compared to Humaneval [9] to prove their validity [13], [15], [49], [50]. However, since the length of code to be generated

<p><b>Before</b></p> <pre>double calculate_area(double radius){     double pi = 3.14159;     double area = pi * radius * radius;     return area; }</pre>	<p><b>Before</b></p> <pre>bool is_even_number(int num) {     if (num % 2 == 0)         return true;     else         return false; }</pre>
<p><b>After</b></p> <pre>double calculate_area(double r){     double b = 3.14159;     double c = b * r * r;     return r; }</pre>	<p><b>After</b></p> <pre>bool is_even_number(int num) {     return !(num % 2 == 0); }</pre>

Fig. 2: Examples of code obfuscation.

by these benchmark tests does not match the distribution of code lengths in Humaneval, this comparison is inherently unfair. It therefore does not accurately reflect the validity of the benchmark tests.

### C. Motivation of Using Code Obfuscation

As LLMs are continuously trained and released, traditional datasets would constantly be learned and trained by these models. Therefore, the timeliness of code may be a crucial factor when testing the generation capacity of LLM. In other words, code may become easier to generate because it already exists (or similar) in the training set.

To validate our conjecture, we conduct a pilot study to check whether the timeliness of code will affect the result of LLM generation capability. We use GPT-3.5-turbo (released at 2023.06.13) and GPT 4.0-preview (released at 2023.11.06) to do code competition problems from LeetCode [51]. We collect the problems from 2018.09 to 2018.11 (90 problems) and the problems from 2023.11 to 2024.01 (127 problems) as our test data. The later 127 problems came out after the release of two models, which were theoretically not presented in the model’s training data.

Table I shows the results. We find that the pass rates of the early 90 problems are much higher than later problems. Due to the fact that code competition websites contain problems of varying difficulty levels, and there is no deliberate increase of the difficulty of new problems, the reason is that these older problems appeared in the model’s training dataset so that models can handle them more easily.

Relying solely on the latest code as a dataset is not a sustainable approach in the long run (as models will continuously train and evolve), and due to the existence of code reuse, even code written after the training cutoff date may still be similar to code in the training set. During the construction of our dataset, to mitigate the impact that LLMs may have seen the code, we have drawn inspiration from code obfuscation techniques [52]–[54], which are originally used for making applications difficult to be decompiled or disassembled. Fig. 2 illustrates two examples of code obfuscation (changing the variable names and changing the code implementation of the same logic). After obfuscation, the code will become different from the training set while maintaining its functionality and output.

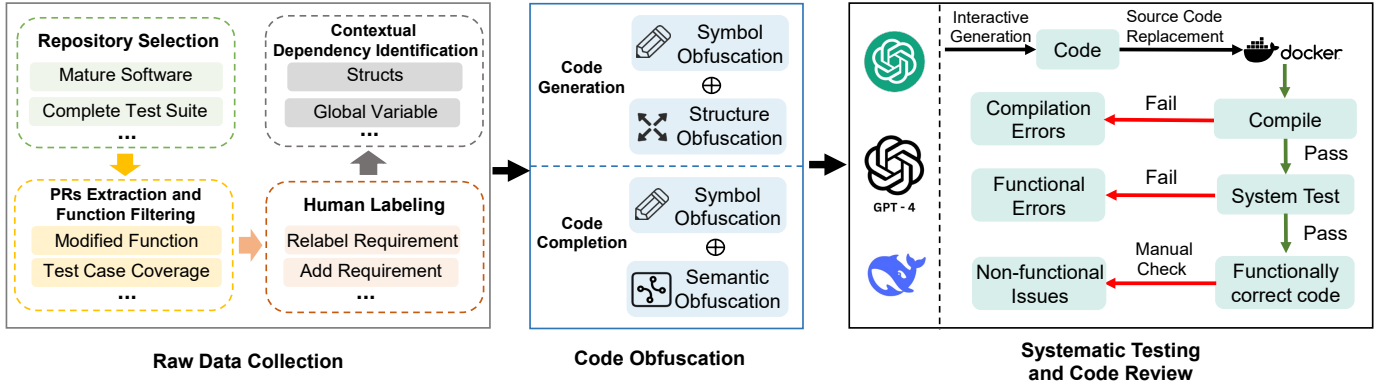


Fig. 3: Workflow of dataset construction and testing

### III. BENCHMARK CONSTRUCTION AND TESTING

In this section, we describe the approach of building and testing OBFUSEVAL to show the capability of LLM, and the workflow is shown in Fig. 3. The building process mainly includes two phases: collecting raw data from open-source projects (Section III-A), and obfuscating code (Section III-B). After that, we construct a testing framework to evaluate the effectiveness of code generation and code completion by the models on OBFUSEVAL (Section III-C).

#### A. Raw Data Collection

To construct OBFUSEVAL, we first need to collect the raw data. The raw data includes functions modified after the large model training data cutoff date from real-world open source projects, as well as the context information these functions depend on within the project. Overall, we divide the raw data collection process into two parts: function collection and contextual dependency provision.

1) *Functions Collection*: Function collection includes three main steps:

**Step 1: Repository Selection.** The existing dataset mainly covers Java and Python projects. To demonstrate the code generation capabilities of the large language models in other programming languages, we chose C projects to build OBFUSEVAL. We set two conditions to filter repositories from GitHub: 1) The repository should be mature and well-maintained; 2) The project should have a comprehensive test suite for systematic testing. Finally, we selected five projects [55]–[59], with over 20k average stars.

**Step 2: PRs Extraction and Function Filtering.** We selected merged PRs from the chosen repositories and extract functions that meet the task criteria. Specifically, we extracted PRs and filter functions based on the following four criteria: 1) The PRs are merged after a certain time point to coincide with the training data cutoff date of specific models we tested; (more discussion in Section IV-A2); 2) The PRs modify the repository’s test files to verify the modified code’s functional correctness; 3) The functions are modified in the PR to ensure that LLM had not seen modified code in previous code base; 4) The functions are covered by the test suites to ensure that the functionality of the functions is effectively verified.

TABLE II: The statics of PRs Extraction and Function Filtering.

Software	Merged PRs	PRs with Tests Modified	Modified Functions	Test-Covered Functions
redis	740	114	3,142	681
libvips	236	14	1,285	203
lvgl	1,718	36	1,447	303
libgit2	118	17	618	78
fluent	160	46	419	89
Total	2,972	227	6,911	1,354

Based on the above criteria, we finally selected 1,354 functions for constructing code generation and code completion tasks and the statistical results are shown in Table II.

**Step 3: Human Labeling.** In this step, we manually provide functional descriptions for each test-covered function. Specifically, we assembled a team of seven senior software engineers, each with at least five years of C programming experience. The team is responsible for rewriting the existing functional descriptions and providing manually written descriptions for functions without descriptions, aiming to reduce the model’s dependence on the original functional descriptions encountered during the pre-training phase. During this process, we implemented a double-check mechanism. When any two engineers have a disagreement, a third engineer is brought in to discuss and reach a final consensus together.

2) *Contextual Dependency Provision*: Previous work [13] has shown that more than 70% of the functions depend on other contextual information in the project, therefore, the inability to provide dependencies can lead to a significant decline in the generative capabilities of large language models. However, accurately providing the dependencies required for code generation is extremely difficult and does not align with real-world development scenarios. Consequently, we adopt a conservative approach to providing dependencies.

We first use syntax tree analysis to identify and collect all relevant contextual dependencies in the code. we extract dependencies from project files, including the names of functions, declarations, function bodies, global variables, structures, macros, as well as function comments. Next, we

TABLE III: The composition of OBFUSEVAL

Soft.	Original Functions	Symbol Obfuscation Functions	Structure Obfuscation Functions	Semantic Obfuscation Functions	Symbol + Structure Obfuscation Functions	Symbol + Semantic Obfuscation Functions
redis	681	681	215	106	215	106
libvips	203	203	58	17	58	17
lvgl	303	303	115	15	115	15
libgit2	78	78	32	10	32	10
fluent	89	89	30	11	30	11
Total	1,354	1,354	450	159	450	159

compile the code to obtain the LLVM IR intermediate code representation of the files. By matching keywords in the IR syntax (such as "call" to indicate a function invocation), we traverse the IR files to acquire the names of those dependencies. By cross-referencing the results from the first step, we can obtain the contextual information of the target function. We also provide similar but different dependencies for each contextual dependency to simulate the disturbances caused by irrelevant information in the actual development process.

### B. Code Obfuscation

Despite selecting code from the project revision history that was modified after the training time of the large model, we also applied additional code Obfuscation techniques to the dataset to enhance protection against "code leakage" and ensure the applicability of our benchmarks in future releases of the large model. Fig. 4 shows the example process of Obfuscating the dataset with three strategies. To objectively evaluate the effectiveness of the LLM when dealing with obfuscated code, we constructed code generation and code completion scenarios, considering the code completion task as a subtask of the code generation task. We apply symbol obfuscation and structure obfuscation strategies in code generation scenarios and symbol obfuscation and semantic obfuscation strategies in code completion scenarios (the strategies can be used in combination).

Distinguishing different obfuscation strategies for various task scenarios is due to the fact that only in the code completion scenario can semantically obfuscated code fragments be retained and a LLM be required to generate complete code. This is to test the LLM's true generation capability when faced with semantically obfuscated code. We do not fully integrate the three types of obfuscation together because different obfuscation strategies are suitable for different types of code. Next, we will discuss in detail the effects and practice of each obfuscation.

1) **Symbol Obfuscation:** We first use a comprehensive symbol obfuscation strategy to not only rewrite the functional descriptions of the functions but also perform thorough identifier rewrites for all meaningful identifiers in the target code and the provided context. This means that all identifiers for functions, variables, class names, etc. in the code are obfuscated regardless of the context in which they appear in the source code. We use NLTK [60] to do the word segmentation and replacement. This strategy is designed at

the token-level to change the LLM's familiarity to the target code.

2) **Structure Obfuscation:** After symbol obfuscation, we further change the code structure automatically using a structure obfuscation strategy. In this stage, we employ the strategy to adjust and integrate the calling structure of the target function so that the execution path and organization of the function are changed. Specifically, we use LLVM [61] to unfold and integrate the functions that are called in the objective function to change the structure of the code. We extract all the called functions and their implementations within the target function based on the context, and then utilize the abstract syntax tree (AST) to handle parameter passing and automatically unfold the called functions.

3) **Semantic Obfuscation:** In semantic obfuscation, we meticulously rewrite code snippets within functions to ensure that the new code is semantically equivalent to the original, yet the implementation logic differs. The goal of this process is to maintain the functional consistency while introducing a new implementation method, effectively obfuscating the code at the method-level. Through semantic transformation, we ensure that the code can still achieve the same functionality, but for the model, its generative logic is completely different from the original code. This semantic obfuscation provides a more challenging task to test the code understanding and generative capabilities of LLM. Since this obfuscation method relies on specific semantics of the code, the current approach involves manually rewriting the code and conducting a double-check. We highly recommend that future research should delve deeper into semantic obfuscation strategies and design templates to automate this process.

Through raw data collection and code obfuscation, we constructed the OBFUSEVAL, which is shown in Table III. Apart from symbol obfuscation, we did not apply structure and semantic obfuscation to all the original data. This is partly because the characteristics of the code may be suitable for specific obfuscation methods (such as nested structures), and partly due to the cost of manual inspection. The code examples from Redis are more numerous and regular, so we applied more semantic obfuscation to them. Future research could design automated code obfuscation framework for obfuscation and inspection.

### C. Systematic Testing and Code Review

To evaluate the LLM's code generation capabilities on the OBFUSEVAL and the effectiveness of our code obfus-



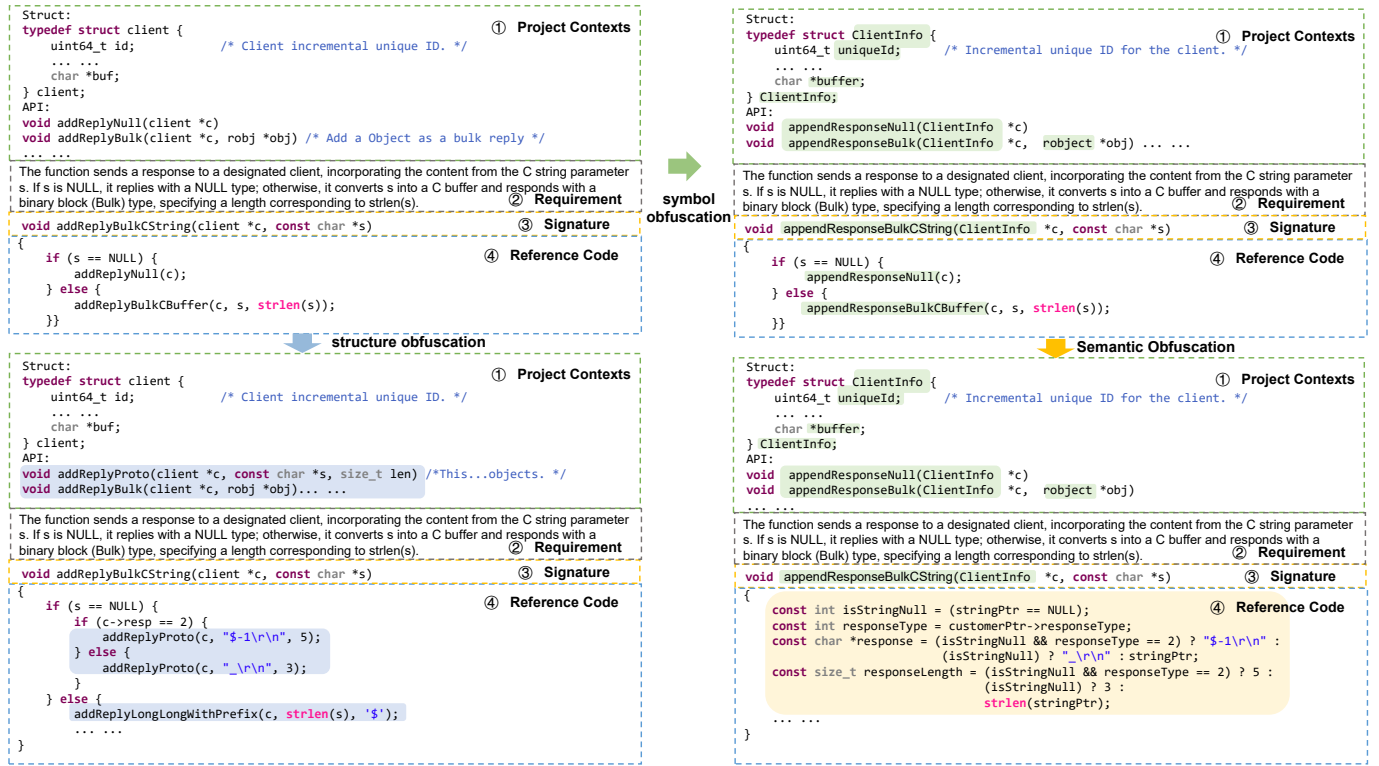


Fig. 4: An example of code obfuscation process

cation methods, we designed an automated code execution and verification platform. The platform is built on Docker images, providing an isolated sandbox environment to ensure that the tested codes do not interfere with each other. The evaluation process of generated code mainly includes two parts: systematic testing and code review.

1) *Systematic Testing*: We use the systematic testing process to evaluate the model and the dataset. We utilize compilation checks and official test suites to detect syntax and semantic errors. We construct the prompt to guide the large model in code generation scenarios and code completion scenarios. The composition of the prompt is detailed below:

- **Instruction**: We provide explicit instructions to guide the LLMs to generate code related to the software.
- **Context**: We provide detailed context information, including structs, macros, functions, global variables, etc. These contexts include both the necessary dependencies for implementing the target function and the context that is irrelevant to the target function.
- **Function description**: We provide a functional description of the target function to guide LLMs in generating code. Based on this description, we provide only the declaration of the target function for **code generation scenarios** and partial code implementation details of the target function for **code completion scenarios**.

Fig. 5 shows an example of the prompt for the code generation scenario. We provide functional descriptions in the [prompt input] to guide the large model's code generation. After obtaining the code generated by the model, we integrate

**Instruct:**

From now on, you play the role of the C code generator. You can generate the corresponding function code according to the function description provided by the user. Please do not return anything other than the target code. Don't return anything other than the function code. The process is as follows:

[prompt-input]

[output]

**Context:**

Here are some function context details you may need to know when writing objective function for the project:

Functions may be used:

```
void addReplyNull(client *c)
```

... ..

Structs may be used:

... ..

Macros may be used:

... ..

Global variables may be used:

... ..

[prompt-input]

This is objective Function Description :

This function sends a reply to the specified ... type reply with a length of strlen(s).

This is the declaration of the objective function:

```
void addReplyBulkCString(client *c, const char *s)
```

[output]

Fig. 5: Example of prompt for code generation scenarios

it into the software for compilation and system testing. If errors occur, we separately record compilation errors and test errors, then analyze the error information. For code that passes the tests, we will conduct a manual code review.

2) *Code Review*: We used compilation checks and test suites to evaluate the syntactic correctness and functional correctness of the generated code, respectively. However,

TABLE IV: **The performance of LLMs in code generation scenarios.** "Original" means the pass rate of all raw code tasks. "Original(Structure)" means the pass rate of those raw code tasks which later perform structure obfuscation. All the numbers have omitted the percentage sign.

Software	Model	Original		Symbol		Original (Structure)		Structure		Symbol+Structure	
		CPR	TPR	CPR	TPR	CPR	TPR	CPR	TPR	CPR	TPR
redis	GPT3.5	38.2	11.9	19.0 ↓	9.7 ↓	44.7	10.7	36.3 ↓	6.5 ↓	29.3 ↓	2.9 ↓
	GPT4-1106	37.0	17.6	31.7 ↓	17.1 ↓	34.4	13.9	36.5 ↑	7.0 ↓	19.6 ↓	4.7 ↓
	GPT4-0125	39.9	20.4	31.8 ↓	17.3 ↓	53.0	13.9	34.9 ↓	9.8 ↓	19.5 ↓	2.8 ↓
	DeepSeek	39.6	7.2	28.8 ↓	11.5 ↑	47.4	5.1	32.1 ↓	5.1 =	22.4 ↓	4.7 ↓
	<b>Average</b>	38.7	14.3	27.8 ↓28.2%	13.9 ↓2.8%	44.9	10.9	35.0 ↓22.0%	7.1 ↓34.9%	22.7 ↓49.4%	3.8 ↓65.1%
libvips	GPT3.5	58.6	18.2	44.8 ↓	18.7 ↑	70.7	20.7	74.1 ↑	20.7 =	46.6 ↓	12.1 ↓
	GPT4-1106	42.9	21.7	32.5 ↓	19.2 ↓	44.8	15.5	44.8 =	20.7 ↑	27.6 ↓	13.8 ↓
	GPT4-0125	43.8	25.6	41.4 ↓	22.2 ↓	51.7	27.6	50.0 ↓	24.1 ↓	48.3 ↓	19.4 ↓
	DeepSeek	50.8	29.6	41.3 ↓	24.6 ↓	53.5	25.9	56.9 ↑	22.4 ↓	44.8 ↓	15.5 ↓
	<b>Average</b>	49.0	23.8	40.0 ↓18.4%	21.2 ↓10.9%	55.2	22.4	56.5 ↑2.4%	22.0 ↓1.8%	41.8 ↓24.3%	15.2 ↓32.1%
lvgl	GPT3.5	36.7	19.8	35.8 ↓	17.5 ↓	27.3	11.6	27.0 ↓	6.9 ↓	13.9 ↓	6.9 ↓
	GPT4-1106	38.9	26.7	39.7 ↑	24.8 ↓	22.6	13.9	23.5 ↑	12.1 ↓	16.5 ↓	10.4 ↓
	GPT4-0125	46.2	31.0	43.3 ↓	28.1 ↓	28.7	15.7	27.0 ↓	11.3 ↓	20.0 ↓	11.3 ↓
	DeepSeek	44.2	30.4	42.6 ↓	28.4 ↓	29.6	14.8	28.7 ↓	9.6 ↓	17.4 ↓	7.0 ↓
	<b>Average</b>	41.5	27.0	40.4 ↓2.7%	24.7 ↓8.5%	27.1	14.0	26.6 ↓1.8%	10.0 ↓28.6%	17.0 ↓37.3%	8.9 ↓36.4%
libgits	GPT3.5	14.1	14.1	11.5 ↓	7.7 ↓	13.1	13.1	12.5 ↓	12.5 ↓	6.2 ↓	3.1 ↓
	GPT4-1106	38.5	23.1	18.0 ↓	9.0 ↓	31.3	25.0	12.5 ↓	12.5 ↓	0.0 ↓	0.0 ↓
	GPT4-0125	39.7	20.5	12.8 ↓	6.4 ↓	34.4	18.8	12.5 ↓	12.5 ↓	0.0 ↓	0.0 ↓
	DeepSeek	23.1	21.8	14.1 ↓	12.8 ↓	18.8	15.6	15.6 ↓	15.6 =	18.6 ↓	9.2 ↓
	<b>Average</b>	28.9	19.9	14.1 ↓51.2%	9.0 ↓54.8%	24.4	18.1	13.3 ↓45.5%	13.3 ↓26.5%	6.2 ↓74.6%	3.1 ↓82.9%
fluent	GPT3.5	27.0	19.1	18.0 ↓	9.0 ↓	33.3	30.0	23.3 ↓	10.0 ↓	30 ↓	3.3 ↓
	GPT4-1106	25.3	20.2	12.4 ↓	10.1 ↓	30.0	26.7	13.3 ↓	10.0 ↓	0.0 ↓	0.0 ↓
	GPT4-0125	34.8	29.2	15.7 ↓	12.4 ↓	43.0	33.3	20.0 ↓	10.0 ↓	16.7 ↓	6.7 ↓
	DeepSeek	19.1	14.6	14.6 ↓	9.0 ↓	13.3	16.7	13.3 =	10.0 ↓	10.0 ↓	3.3 ↓
	<b>Average</b>	26.6	20.8	15.2 ↓42.9%	10.1 ↓51.4%	29.9	26.7	17.5 ↓41.5%	10.0 ↓62.5%	14.2 ↓52.5%	3.3 ↓87.6%
<b>Average</b>		36.9	21.1	27.5 ↓25.5%	15.8 ↓25.1%	36.3	18.4	29.7 ↓18.2%	12.5 ↓32.1%	20.4 ↓43.8%	6.9 ↓62.5%

<sup>1</sup> ↑ / ↓ means that the pass rate has increased / decreased by less than 30% (ratio). ↑ / ↓ means that the pass rate has increased / decreased by more than 30% (ratio).

<sup>2</sup> The change magnitudes are calculated by comparing to "Original" and the "Original(Structure)".

through manual inspection, we found that even the code was functionally correct, i.e., it was able to pass the tests, there were still some non-functional code quality issues. These problems may include deficiencies in code efficiency, code robustness, etc. Therefore, we manually analyzed these non-functional code quality issues (Section IV-B3).

#### IV. EVALUATION

In this section, we describe our experimental setup and the evaluation results of four LLMs on our dataset. Our evaluation focuses primarily on the performance of LLMs on our dataset, and whether code obfuscation can further reveal the true capabilities of these LLMs.

##### A. Evaluation Setup

1) **Model selection:** We chose a general-purpose large language model, ChatGPT, and a code-focused large language model, DeepSeek. Both of them are mature and widely-used LLMs. For ChatGPT, we use the "gpt-3.5-turbo-1106" [62], "gpt-4-turbo-1106" [63], and "gpt-4-turbo-0125" [63] in our experiments. For DeepSeek, we use DeepSeek-coder-v2 [64] with the default settings. We use default value for LLM's parameters when generating code.

2) **Raw data selection:** Note that the training datasets for gpt-4-turbo-1106 were finalized as of April 2023 (gpt-3.5-turbo-1106 is also before that time). We selected the

code starting from May 2023 to Dec 2023 as the original data. Therefore, for gpt-3.5-turbo-1106 and gpt-4-turbo-1106, these codes were not included in the training set, while for gpt-4-turbo-0125 and DeepSeek-coder-v2, they might have encountered some of these code snippets during training. By doing so, we not only ensure a balanced distribution in our dataset (with both seen and unseen data), but also objectively and authentically demonstrate the generative capabilities of LLMs across different types of code.

3) **Evaluation Metrics:** To assess the correctness of the generated code snippet, we employed two key performance metrics to measure the code generation capabilities of the LLMs in real-world development scenarios: **Compile Pass Rate (CPR)** and **Test Pass Rate (TPR)**. We first replace the original function with the function generated by the LLMs and then compile the software. After that, we perform system tests associated with that function.

CPR and TPR are representative metrics that can illustrate generated code that passes compilation and tests respectively. CPR demonstrates the basic ability of large language models to generate syntactically correct code, while TPR reflects the capability of large language models to understand and correctly generate complex functional code. Specifically, TPR can demonstrate whether the LLM can be used directly in production scenarios. Both CPR and TPR use pass@5 rate to eliminate fluctuations.

TABLE V: **The performance of LLMs in code completion scenarios.** "Original" means the pass rate of all raw code tasks. "Original(Semantic)" means the pass rate of those raw code tasks which later perform semantic obfuscation. All the numbers have omitted the percentage sign.

Software	Model	Original		Symbol		Original (Semantic)		Semantic		Symbol+Semantic	
		CPR	TPR	CPR	TPR	CPR	TPR	CPR	TPR	CPR	TPR
redis	GPT3.5	19.0	9.0	30.9 ↑	11.5 ↑	17.9	13.2	13.2 ↓	7.5 ↓	36.4 ↑	11.2 ↓
	GPT4-1106	41.5	25.4	32.9 ↓	14.2 ↓	51.9	36.8	44.3 ↓	25.5 ↓	39.6 ↓	20.7 ↓
	GPT4-0125	39.3	15.4	36.1 ↓	15.9 ↑	43.4	17.9	38.7 ↓	24.5 ↑	49.1 ↑	25.5 ↑
	DeepSeek	40.6	24.8	30.9 ↓	4.0 ↓	54.7	40.6	56.6 ↑	31.1 ↓	53.8 ↓	32.1 ↓
	<b>Average</b>	35.1	18.7	32.7 ↓6.8%	11.4 ↓39.0%	42.0	27.1	38.2 ↓9.0%	22.2 ↓18.1%	44.7 ↑6.4%	22.4 ↓17.3%
libvips	GPT3.5	19.7	8.4	31.0 ↑	13.3 ↑	41.2	11.8	29.4 ↓	5.9 ↓	41.2 =	10.6 ↓
	GPT4-1106	36.0	22.2	31.5 ↓	18.2 ↓	35.3	23.5	41.1 ↑	23.5 =	35.3 =	17.7 ↓
	GPT4-0125	44.3	27.1	33.5 ↓	22.2 ↓	47.0	29.4	47.0 =	23.5 ↓	29.4 ↓	29.4 =
	DeepSeek	36.9	25.1	37.0 ↑	21.7 ↓	41.1	23.5	41.1 =	23.5 =	35.3 ↓	23.5 =
	<b>Average</b>	34.2	20.7	33.3 ↓2.6%	18.9 ↓8.7%	41.2	22.1	39.7 ↓3.6%	19.1 ↓13.6%	35.3 ↓14.3%	20.3 ↓8.1%
lvg1	GPT3.5	22.4	18.2	27.2 ↑	20.5 ↑	6.7	0.0	6.7 =	0.0 =	6.7 =	0.0 =
	GPT4-1106	39.6	30.0	31.4 ↓	24.1 ↓	20.0	13.3	20.0 =	13.3 =	13.3 ↓	13.3 =
	GPT4-0125	44.2	33.00	36.3 ↓	27.1 ↓	20.0	13.3	20.0 =	13.3 =	13.3 ↓	6.7 ↓
	DeepSeek	35.2	29.4	28.2 ↓	11.9 ↓	20.0	13.3	20.0 =	13.3 =	13.3 ↓	13.3 =
	<b>Average</b>	35.4	27.7	30.8 ↓13.0%	20.9 ↓24.5%	16.7	10.0	16.7 =	10.0 =	11.7 ↓29.9%	8.3 ↓17.0%
libgits	GPT3.5	15.4	12.8	15.4 =	12.8 =	10.0	10.0	10.0 =	10.0 =	20.0 ↑	10.0 =
	GPT4-1106	10.3	10.3	6.4 ↓	6.4 ↓	60.0	30.0	50.0 ↓	30.0 =	20.0 ↓	20.0 ↓
	GPT4-0125	12.8	10.3	7.7 ↓	7.7 ↓	50.0	40.0	50.0 =	30.0 ↓	40.0 ↓	30.0 ↓
	DeepSeek	25.6	21.8	23.0 ↓	19.2 ↓	30.0	30.0	40.0 ↑	20.0 ↓	30.0 =	20.0 ↓
	<b>Average</b>	16.0	13.8	13.1 ↓18.1%	11.5 ↓16.7%	37.5	27.5	37.5 =	22.5 ↓18.2%	27.5 ↓26.7%	20.0 ↓27.3%
fluent	GPT3.5	11.2	7.87	16.9 ↑	11.2 ↑	0.0	0.0	0.0 =	0.0 =	18.2 ↑	0.0 =
	GPT4-1106	25.8	23.6	16.9 ↓	11.2 ↓	9.1	9.1	18.2 ↑	9.1 =	9.1 =	9.1 =
	GPT4-0125	30.3	27.0	19.1 ↓	19.1 ↓	18.2	18.2	18.2 =	9.1 ↓	18.2 =	9.1 ↓
	DeepSeek	21.4	18.0	14.6 ↓	11.2 ↓	18.2	18.2	18.2 =	18.2 =	9.1 ↓	9.1 ↓
	<b>Average</b>	22.2	19.1	17.2 ↓22.5%	13.2 ↓30.9%	11.4	11.4	13.7 ↑20.2%	9.1 ↓20.2%	13.7 ↑20.2%	6.8 ↓40.4%
<b>Average</b>		28.6	20.0	25.4 ↓11.2%	15.2 ↓24.0%	29.7	19.6	29.1 ↓2.0%	16.6 ↓15.3%	26.6 ↓10.4%	15.6 ↓20.4%

<sup>1</sup> ↑ / ↓ means that the pass rate has increased / decreased by less than 30% (ratio) . ↑ / ↓ means that the pass rate has increased / decreased by more than 30% (ratio) .

<sup>2</sup> The change magnitudes are calculated by comparing to "Original" and the "Original(Semantic)".

## B. Results and Analysis

To evaluate the performance of large models on our dataset, we conduct experiments on four large models using OBFUSEVAL. Specifically, we explore the effectiveness of code obfuscation and the various among different obfuscation strategies. We investigate the following three research questions:

- **RQ1:** How effective are large language models in generating code on our datasets?
- **RQ2:** How does code obfuscation further reveal the capabilities of LLMs, and how effective are different obfuscation strategies?
- **RQ3:** What are the issues hidden in LLM-generated code?

1) **RQ1: How effective are large language models in generating code on our datasets:** The model usually performs better on the code similar to training data. Thus, we collected code data after model training and utilized code obfuscation methods to further eliminate code leakage. In this research question, we conduct experiments with several widely used and proven effective large language models on our collected dataset. The overall results are presented in Table IV and Table V.

The CPRs are among 16.0% to 49.0% on original raw code, with an average of 32.8%, which is significantly lower than the LLM's performance on traditional datasets (e.g., humaneval). The average TPRs are 21.1% and 20.0% on original code

generation and completion. After we applied different levels of code obfuscation, this result dropped to 6.9% and 15.6%. This indicates that the code generated by LLMs is difficult to pass the official tests of software and be directly used in production environments. Developers still need to manually fix and adjust the code.



**Conclusion for RQ1:** Large Language Models still fall significantly short of meeting the requirements for unfamiliar code generation/completion tasks in real-world production environments. The basic pass rate of compilation remains a significant bottleneck. Even if it passes the compilation, it may still not meet the actual functional requirements of the software.

2) **RQ2: How does code obfuscation further reveal the capabilities of LLMs, and how effective are different obfuscation strategies:** We use three obfuscation strategies (Section III-B) to modify the collected raw code to minimize the possibility that the code might resemble the training data. We specifically compare the TPR before and after different code obfuscations to evaluate the syntactic and functional correctness of generated code. We try them separately and also their combinations. The results are shown in Fig. 6 (detailed data can be found in Table IV and Table V).

**Overall effect.** All obfuscation strategies have reduced the TPR of code generation by LLMs. Since we did not deliberately increase the difficulty and complexity of the code



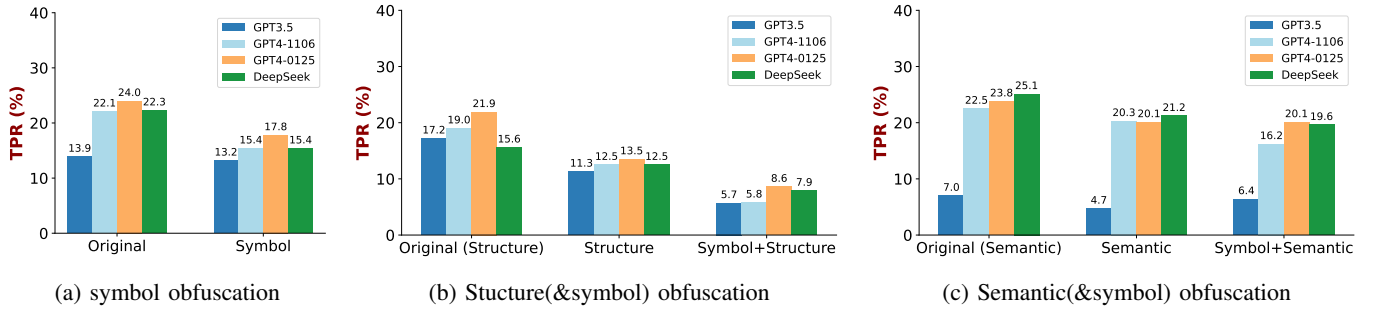


Fig. 6: TPR under different code obfuscation strategies. **After introducing various degrees of obfuscation, the rankings of four LLMS tend to be stabilize and consistent.**

in code obfuscation (discussed in Section V), code obfuscation can make the tested code less familiar to large models. Even though the decline may not be particularly significant, it represents the trend of the LLMs’ capabilities, and if applied in a large-scale production environment, it could still have a considerable impact.

As for tested LLMs, GPT4-1106, GPT4-0125 and DeepSeek Code are obviously better than GPT3.5. However, the ranking of their capability on the Original code is inconsistent. After obfuscation, their ranking becomes more stable (GPT4-0125>DeepSeek Code>GPT4-1106>GPT3.5). This further proves that using code obfuscation can more accurately demonstrate the capabilities of the models.

**Strategy comparison.** In this paper, we intuitively determine the effectiveness of code obfuscation strategies by the TPR decrease ratio comparing to the Original code tasks (eliminate the inflated capabilities of the large model). Symbol and structure obfuscation can be very effective with the average TPR decrease ratio of 24.6% and 32.1%. Semantic obfuscation is not that effective, with an average decrease ratio 15.3%. This is mainly because our current semantic obfuscation strategies are still relatively simple (using some heuristics to rewrite code manually), without making in-depth modifications to the code. We also find that the use of mixed strategies can lead to a further decrease in TPR (Symbol + Structure can have an average decrease ratio of 62.5%), demonstrating the effectiveness of mixed strategies.

Although different obfuscation methods have different effectiveness, we still recommend that future researchers should try various obfuscation strategies and their combinations, which can not only ensure the fairness of code obfuscation but also explore the performance of LLMs on more types of code implementations.



**Conclusion for RQ2:** All obfuscation strategies can help eliminate the inflated code generation capabilities of LLMs. symbol and structure obfuscation are particularly effective. Future research could further explore more sophisticated semantic obfuscation methods and additional strategies, and automate the entire obfuscation process efficiently.

3) **RQ3: What are the issues hidden in LLM-generated code:** To enable developers to better utilize the code generated by large language models for downstream development tasks,

we further analyzed common issues hidden in the generated code that may affect development tasks. Due to the large size of the OBFUSEVAL, we employed stratified sampling for manual code review. The sample size was calculated through the finite population correction, ensuring our sample accurately represents the dataset. We set the confidence level to 95% and the margin of error to 5%, which are standard statistical thresholds. Through this method, we extracted 299 pieces of LLM-generated code from five software and identified the following three categories of issues:

- **Syntax errors:** Codes lead to compilation errors.
- **Functional errors:** Codes fail to meet the functional requirements of development (fail the official tests).
- **Non-functional code quality issues:** Codes potentially causing performance and reliability issues.

Due to the target software being from diverse domains, the functionalities and the code semantics are various, so we mainly focus on **syntax errors** and **non-functional code quality issues**.

**Syntax Errors.** We refer to the LLM-generated code that causes compilation errors in the software as syntax error code. Such errors significantly impact development efficiency. We extract error logs and manually analyze the code that failed to compile, ultimately categorizing the syntax errors into 5 major categories and 21 subcategories, as shown in Table VI.

In summary, large language models perform worst in handling function and type declarations, particularly implicit function declarations and type conflicts. And LLMs perform poorly in generating code related to structures, often resulting in issues such as accessing non-existent members. These errors may be due to the presence of similarly named functions or structures in the training data of the LLMs, leading to the use of functions or structures outside the provided context in code generation tasks. We recommend that developers focus on checking external dependencies such as called functions and structures when using LLMs to generate code. Additionally, LLMs often perform poorly when handling code involving pointers, leading to errors such as converting integers to pointers without casting, and incompatible pointer types.

**Non-functional code quality issues.** For LLM-generated codes that have passed systematic testing, we conducted a manual code quality review and found that the code often

TABLE VI: Syntax errors of LLM-generated code

Category	Proportion
<b>Function and Type Declaration Errors</b>	<b>45.24%</b>
Implicit declaration of function	30.56%
Type conflict	13.35%
API parameter count mismatch	1.02%
Undeclared type	0.31%
<b>Data Structure and Member Access Errors</b>	<b>26.70%</b>
Non-existent structure member	19.87%
Misuse of structure pointer	6.60%
Use $\rightarrow$ operator to access an integer member	0.23%
<b>Type Conversion and Assignment Errors</b>	<b>12.82%</b>
Making a pointer from an integer without a cast	8.57%
Incompatible pointer type	1.25%
Incompatible type assignment	1.50%
Redefinition	1.50%
<b>Scope and Definition Errors</b>	<b>1.89%</b>
Conflict between static and non-static declarations	1.50%
Incorrect access to structure or union member	0.39%
<b>Other Syntax Errors</b>	<b>13.35%</b>
Lvalue required as the left operand of assignment	6.60%
Incorrect use of array, pointer, or vector	1.73%
Assignment to expression with array type	0.63%
Incorrect use of parentheses	0.63%
Invalid binary operands	0.55%
Expected expression error	0.47%
Array subscript is not an integer	0.23%
Subscripted value is pointer to function	0.23%
Others	2.28%

had non-functional quality issues such as poor performance, and security vulnerabilities. These issues do not directly affect the normal execution of the code, but they can pose potential threats to the software. For example, poor performance may cause system response time delays, affecting user experience and overall performance; security vulnerabilities may be exploited by malicious attackers, leading to data breaches or system crashes. We have summarized common non-functional code quality issues into the following three categories:

- **Resource Management:** In LLM-generated code, resource management code is often inadequate, especially in terms of memory or file management. For example, LLM-generated code often does not free dynamically allocated memory at the end of the program. Such code can pass compilation and testing, but it may lead to potential memory leaks, resulting in software performance issues.
- **Code Efficiency:** Fig. 7(a) shows an example of low code efficiency. *Wtiff\_pack2tiff* is an image data conversion function in the *libvips* software. In the original code, the function adopts corresponding processing methods according to different image encoding formats, while the LLM-generated code uses an inefficient loop to process each pixel, modifying and copying pixel by pixel. Although the LLM-generated code can meet functionality, its performance is far inferior to the original code, affecting the overall performance of the software.
- **Code Robustness:** The robustness of the generated code is not guaranteed, including missing error checking code, incomplete error handling and inadequate feedback message.

```

/* Original code */
Void wtiff_pack2tiff(Wtiff *wtiff, VipsRegion *in,VipsRect *area,
VipsPel *q){ // Different condition -> Different method
for(int y = area->top;y < RECT_BOTTOM( area );y++) {
VipsPel *p = (VipsPel *) REGION_ADDR(in,area->left,y);
if(wtiff->ready->Coding == CODING_LABQ)
LabQ2LabC( q, p, area->width );
else if ..... // Omit multiple branches
else
memcpy(q,p,area->width *IMAGE_SIZEOF_PEL(wtiff->ready));
.....
}

```

(a) An example of low efficiency code.

```

/* Original code */
Void clusterUpdateMyselfAnnouncedPorts(void) {
if (!myself) return; // Error Handling
deriveAnnouncePorts(&myself->port,&myself->pport,
&myself->cport);
}


/* LLM-generated code */
Void clusterUpdateMyselfAnnouncedPorts(void) {
myself->port = server.cluster_announce_port;
myself->cport = server.cluster_announce_bus_port;
myself->pport = server.cluster_announce_tls_port;
}

```

(b) An example of missing error handling.

Fig. 7: Examples of non-functional code quality issue.

Fig. 7(b) shows an example of missing error handling. In the original code, the program first checks if the *myself* pointer is NULL to prevent null pointer reference. However, the LLM-generated code does not perform null pointer checks, which could lead to accessing the pointer when it is uninitialized or freed, causing software crashes.

 **Conclusion for RQ3:** Function and Type Declaration Errors are the most common syntax error of LLM generated code. While, even if the code passes compilation and system testing, it may still have non-functional issues which will negatively affect software reliability and performance. Future research should establish more comprehensive evaluation metrics to evaluate the quality of code generated by LLMs.

## V. THREATS TO VALIDITY

**Raw data collection.** To simulate real-world development scenarios, we select mature system software from the real world as the target software for our research. We chose these software systems because they are widely used, have a rich development history, and contain mature test suites. We believe our study is representative, although some results may not apply to all kinds of software and all kinds of code language. There are other human efforts evolved in the data collection process (e.g., description rewriting). To minimize the impact of human error, we organize a team of seven senior software engineers, each with at least five years of experience in C programming. Additionally, we conduct a double-check progress in each step.

**Code complexity.** Obfuscation strategies can introduce changes in code complexity, primarily in structural and semantic obfuscation. For example, function inlining in structural obfuscation may lead to an increase in the number of lines of code, potentially affecting the generation capabilities of large models. In practice, we did not intentionally increase the complexity of the code in any obfuscation method; our guiding principle was to ensure that the semantics of the code remained the same before and after obfuscation.

**Testing process.** Due to the presence of flaky tests, the testing environment of real software projects can be unstable. Even correct code may fail tests due to contextual or environmental issues. Therefore, for each case, we run tests at least five times to mitigate intermediate test results. Official test suites may not comprehensively test the correctness of functionality, but this represents the best efforts, allowing for a better assessment of whether the code meets development requirements.

## VI. CONCLUSION

Accurately assessing the code generation capabilities of LLMs is crucial for their evaluation and improvement. While existing works have constructed datasets to gauge these capabilities, three main gaps persist in objectively evaluating LLMs' real potential: the exposure of target code, case timeliness, and dependency availability. These gaps arise because the code in current datasets may have been exposed during the training phase of LLMs, and the continuous training and development of LLMs severely compromise their timeliness. To address the problem, this paper adopts the concept of code obfuscation, altering code at various levels while preserving its functionality and output. We developed a code-obfuscation-based benchmark, OBFUSEVAL, by collecting 1,354 raw cases from five real-world projects, which include function descriptions and code. We then obfuscated descriptions, code, and context dependencies using a three-level strategy (symbol, structure, and semantic). Evaluating four LLMs on OBFUSEVAL and comparing the effectiveness of different obfuscation strategies, we found that after obfuscation, the average test pass rate can decreased by 15.3%-62.5%.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This research was funded by NSFC No.62272473, the Science and Technology Innovation Program of Hunan Province No.2023RC1001, NSFC (No.U2441238, No.62202474) and National University of Defense Technology Research Project No.ZK24-01.

## REFERENCES

- [1] M. Liu, J. Wang, T. Lin, Q. Ma, Z. Fang, and Y. Wu, "An empirical study of the code generation of safety-critical software using llms," *Applied Sciences*, vol. 14, no. 3, p. 1046, 2024.
- [2] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.
- [3] F. Lin, D. J. Kim *et al.*, "When llm-based code generation meets the software development process," *arXiv preprint arXiv:2403.15852*, 2024.
- [4] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. Grossman, "How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment," in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 2023, pp. 1–12.
- [5] H. Kozirolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani, "Llm-based and retrieval-augmented control code generation," in *Proc. 1st Int. Workshop on Large Language Models for Coffice (LLM4Code)* at ICSE, vol. 2024, 2024.
- [6] 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.
- [7] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.
- [8] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," *arXiv preprint arXiv:2408.06450*, 2024.
- [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] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multi-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.
- [11] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei, "Aixbench: A code generation benchmark dataset," *arXiv preprint arXiv:2206.13179*, 2022.
- [12] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18319–18345.
- [13] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.
- [15] J. Li, G. Li, Y. Zhao, Y. Li, Z. Jin, H. Zhu, H. Liu, K. Liu, L. Wang, Z. Fang *et al.*, "Deveval: Evaluating code generation in practical software projects," *arXiv preprint arXiv:2401.06401*, 2024.
- [16] Q. Zhu, J. Cao, Y. Lu, H. Lin, X. Han, L. Sun, and S.-C. Cheung, "Domaineval: An auto-constructed benchmark for multi-domain code generation," *arXiv preprint arXiv:2408.13204*, 2024.
- [17] "Github," <https://github.com/>, 2024.
- [18] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic code generation from design patterns," *IBM systems Journal*, vol. 35, no. 2, pp. 151–171, 1996.
- [19] 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.
- [20] X. Huang, Y. Ma, H. Zhou, Z. Jiang, Y. Zhang, T. Wang, and S. Li, "Towards better multilingual code search through cross-lingual contrastive learning," in *Proceedings of the 14th Asia-Pacific Symposium on Internetwork*, 2023, pp. 22–32.
- [21] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [22] Openai, "gpt-4 technical report," corr, vol. abs/2303.08774, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2303.08774>
- [23] Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang, "Glm: General language model pretraining with autoregressive blank infilling," *arXiv preprint arXiv:2103.10360*, 2021.
- [24] 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.

- [25] Y. Chang, X. Wang, J. Wang, Y. Wu, K. Zhu, H. Chen, L. Yang, X. Yi, C. Wang, Y. Wang *et al.*, “A survey on evaluation of large language models,” *arXiv preprint arXiv:2307.03109*, 2023.
- [26] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao *et al.*, “Pangu-coder2: Boosting large language models for code with ranking feedback,” *arXiv preprint arXiv:2307.14936*, 2023.
- [27] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, “StarCoder 2 and the stack v2: The next generation,” *arXiv preprint arXiv:2402.19173*, 2024.
- [28] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, “Large language models meet nl2code: A survey,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.
- [29] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, “WizardCoder: Empowering code large language models with evol-instruct,” *arXiv preprint arXiv:2306.08568*, 2023.
- [30] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, “Deepseek-coder: When the large language model meets programming—the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [31] Y. Ma, Y. Liu, Y. Yu, Y. Zhang, Y. Jiang, C. Wang, and S. Li, “At which training stage does code data help llms reasoning?” *arXiv preprint arXiv:2309.16298*, 2023.
- [32] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, “Cct5: A code-change-oriented pre-trained model,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1509–1521.
- [33] Y. Dong, G. Li, and Z. Jin, “Codep: grammatical seq2seq model for general-purpose code generation,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 188–198.
- [34] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, “InCoder: A generative model for code infilling and synthesis,” *arXiv preprint arXiv:2204.05999*, 2022.
- [35] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “StarCoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [36] “(2023) instruct-starcoder,” <https://huggingface.co/GeorgiaTechResearchInstitute/starcoder-gpteacher-code-instruct>, 2023.
- [37] “Chatgpt,” <https://chatgpt.com>, 2024.
- [38] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, “SkCoder: A sketch-based approach for automatic code generation,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2124–2135.
- [39] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, “KareCoder: A new knowledge-enriched code generation system,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 270–271.
- [40] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, “Evocodebench: An evolving code generation benchmark aligned with real-world code repositories,” *arXiv preprint arXiv:2404.00599*, 2024.
- [41] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [42] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, “Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x,” *arXiv preprint arXiv:2303.17568*, 2023.
- [43] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, “Competition-level code generation with alphacode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
- [44] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman *et al.*, “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task,” *arXiv preprint arXiv:1809.08887*, 2018.
- [45] F. He, J. Zhai, and M. Pan, “Beyond code generation: Assessing code llm maturity with postconditions,” *arXiv preprint arXiv:2407.14118*, 2024.
- [46] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, “Exploring and evaluating hallucinations in llm-powered code generation,” *arXiv preprint arXiv:2404.00971*, 2024.
- [47] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, A. Alipour, S. Jha, P. Devanbu, and T. Ahmed, “Calibration and correctness of language models for code,” *arXiv preprint arXiv:2402.02047*, 2024.
- [48] J. J. Wu and F. H. Fard, “Benchmarking the communication competence of code generation for llms and llm agent,” *arXiv preprint arXiv:2406.00215*, 2024.
- [49] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation,” *arXiv preprint arXiv:2308.01861*, 2023.
- [50] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, “AceCoder: An effective prompting technique specialized in code generation,” *ACM Transactions on Software Engineering and Methodology*, 2024.
- [51] “LeetCode,” <https://leetcode.cn/>, 2024.
- [52] S. Schrittwieser and S. Katzenbeisser, “Code obfuscation against static and dynamic reverse engineering,” in *Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13*. Springer, 2011, pp. 270–284.
- [53] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, “Protecting software through obfuscation: Can it keep pace with progress in code analysis?” *Acm computing surveys (csur)*, vol. 49, no. 1, pp. 1–37, 2016.
- [54] A. Balakrishnan and C. Schulze, “Code obfuscation literature survey,” *CS701 Construction of compilers*, vol. 19, p. 31, 2005.
- [55] “Redis,” <https://github.com/redis/redis>, 2024.
- [56] “Libgit2,” <https://github.com/libgit2/libgit2>, 2024.
- [57] “Libvips,” <https://github.com/libvips/libvips>, 2024.
- [58] “Fluent,” <https://github.com/fluent/fluent-bit>, 2024.
- [59] “lvgl,” <https://github.com/lvgl/lvgl>, 2024.
- [60] “Nltk,” <https://www.nltk.org/>, 2024.
- [61] “Llvm,” <https://llvm.org/>, 2024.
- [62] “gpt-3.5-turbo,” <https://platform.openai.com/docs/models/gpt-3-5-turbo>, 2024.
- [63] “gpt-4.0-turbo,” <https://platform.openai.com/docs/models/gpt-4o>, 2024.
- [64] “Deepseek-coder-v2,” <https://github.com/deepseek-ai/DeepSeek-Coder-V2>, 2024.