Final Project Report

-How to use ChatGPT to analyze codes efficiently  
by Prompt Engineering

Background:

While I was initially tasked with creating a tool to assess code coverage, I encountered a roadblock. The current tools in the market seemed quite effective, leaving little room for improvement. Amidst my confusion over this dilemma, a friend sought my assistance in debugging his code. Aware of ChatGPT's impressive capabilities, I suggested he consult ChatGPT for solutions. However, he expressed his frustration, having already exhausted his efforts in that direction. Although slightly impatient, I set aside my code coverage project to aid him. This decision ultimately led to a pivotal shift in my perspective on choosing my project topic. As I observed my friend's interaction with GPT, I noticed that he was relying on it to identify potential bugs. This observation sparked a novel idea in my mind – why not explore ways to enhance the efficiency of utilizing ChatGPT in programming environments?

Motivation:

The project's motivation stems from recognizing the limitations of conventional code analysis methods, like Randoop and fuzz testing. These traditional approaches often fall short in efficiency due to three key drawbacks: their inherent randomness, the absence of logical explanations for their findings, and constraints imposed by specific programming languages. This project proposes a novel approach: leveraging ChatGPT as an alternative tool for code analysis. The central question driving this project is not just about integrating ChatGPT into the analysis process, but rather, how to optimize its use to ensure it outperforms traditional methods. We aim to explore strategies and frameworks that can effectively utilize ChatGPT's capabilities, ensuring that it provides a more efficient, logical, and versatile solution for code analysis than its predecessors.

Methodology:

1. General steps:

(a)File Analysis

(b)Prompt Generation

(c)Output generated by LLM.

2. Description: File Analysis is like linting, it leads to understand the class and figure out what is related to the class (e.g., other class or packages). If we are not using the strong-type programming language (such as C++), we only get to know what the type of the data is in runtime. Prompt generation aims to solve this problem.

Implementation:

1. File Analysis:

文本

描述已自动生成

From the screenshot, we can see that the program was analyzed manually, and the picture below is an example of “what is related to the class”.

2. Prompt Generation & Output generated by LLM

(a)

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

This is a super simple prompt, and the screenshot indicates that there is no constrains on the prompt.

Analysis Report:

1.It is Stack Buffer Overflow

2.It is Heap Buffer Overflow

3.We can ignore Error handling for malloc to return null

4.And forth one is also related to the heap buffer overflow attack.

5.The fifth one occurs because ChatGPT does not have the context of our code so we can also ignore this. And so does 6 and 7.

6.The logging issue is a generic output from ChatGPT. It is there for liability reasons.

(b)

图形用户界面, 文本, 应用程序

描述已自动生成

After setting up this constrain, the stack buffer overflow issue will disappear.

Analysis Report:

1. Unchecked Return Value is a new issue, but it can be ignored because ChatGPT does not know what stream\_Seek () is. And this is also what we can add to the prompt if we want to improve it more.

2. Integer Overflow issue and Unchecked User Data issue are related to the previous issue named as “Type mismatch”.

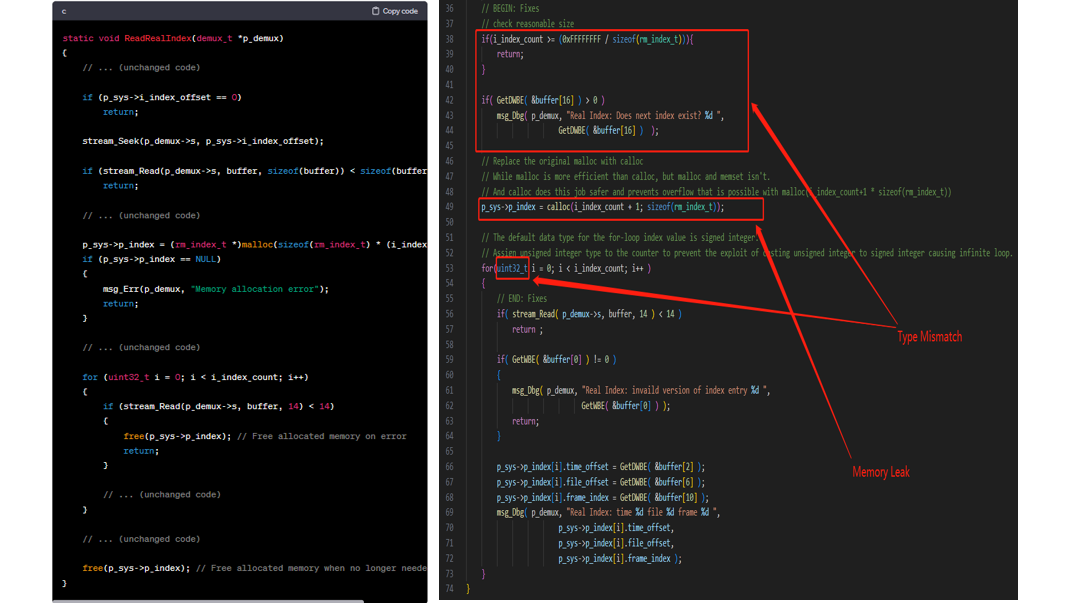
3. Potential Heap Buffer Overflow is same as the previous issue named as Memory Leak.

4. Memory Allocation Failure is like the issue Error Handling.

5. Magic Numbers and Error handling have existed before.

6. Lack of Comment is a friendly remind.

Evaluation:



We finally figured out debugging on this part of program by following ChatGPT’s instructions and improving the prompt generation.

Futural Enhancement:

While the generated codes are correct, it doesn’t check deprecation status from the vendor.

(a) Let’s move on to some high-level programming language like swift.

文本

描述已自动生成

(b) This screenshot indicates that the current method is unstable. The warning shows that early access may occur. This code generated by GPT does not have a correct context of the variable, therefore, the error occurs.

电脑屏幕的截图

描述已自动生成

(c) This is a method with the latest guidance from Apple. In this way, we don’t have to use the deprecated method.

文本

描述已自动生成

Explanation:

Apple's shift towards Swift 6, with its emphasis on strict concurrency-safe coding, comes alongside major updates to iOS that introduce new SDKs. This transition has led Apple to phase out older methodologies in favor of more contemporary approaches. A key feature in Xcode, Apple's Integrated Development Environment (IDE), is the ability to easily identify concurrency and thread-safety issues. This is achieved by enabling strict concurrency checking within the project settings.

A straightforward method to ensure data processing occurs on the main thread involves the use of the @MainActor flag. This flag can be applied either to specific methods or to an entire class. Its role is pivotal in leveraging the compiler's understanding of the code, aiming to operate within defined actor boundaries. However, it's important to note that simply using the @MainActor flag does not automatically make existing code compliant with Swift 6, especially when data within an execution block crosses these actor boundaries.

The concept of a Swift actor, introduced in Swift 5.5, is central to these updates. Swift actors are a concurrency feature designed to manage shared mutable state safely and efficiently. Functioning as a type of object, they can be accessed concurrently across multiple threads. What sets them apart is their ability to ensure that access to their mutable state is serialized and thread-safe, marking a significant advancement in Swift's approach to modern, safe coding practices.

Conclusion:

Our exploration into Prompt Engineering reveals that while it may appear straightforward, as evidenced by our example, it is in fact a multifaceted and intricate process. The effectiveness of Prompt Engineering hinges on the precision and adequacy of the information (or hints) provided to the model. This is crucial for producing outputs that are not just relevant, but also of practical utility. A prime illustration of this is found in the realm of AI-generated imagery. It is evident that the quality of prompts directly influences the resultant image quality. Professional artists, with their detailed and accurate descriptions, can elicit far superior images from AI models compared to novices who might input vague or simplistic sentences and expect detailed outcomes. This disparity underscores the significance of skillful Prompt Engineering in optimizing the capabilities of AI models, a key takeaway from our project findings.