LLMs: Understanding Code Syntax and Semantics for Code Analysis

Wei Ma
Continental-NTU Corporate Lab, Nanyang Technological University
Singapore

Shangqing Liu
Nanyang Technological University
Singapore

Zhihao Lin
Beihang University
China

Wenhan Wang
Nanyang Technological University
Singapore

Qiang Hu
University of Luxembourg
Luxembourg

Cen Zhang
Nanyang Technological University
Singapore

Ye Liu
Nanyang Technological University
Singapore

Li Li
Beihang University
China

Liming Nie
Beihang University
Singapore

Yang Liu
Nanyang Technological University
Singapore

ABSTRACT

Large language models (LLMs) demonstrate significant potential to revolutionize software engineering (SE) by exhibiting outstanding performance in SE tasks such as code and document generation. However, the high reliability and risk control requirements in software engineering raise concerns about the lack of interpretability of LLMs. To address this concern, we conducted a study to evaluate the capabilities of LLMs and their limitations for code analysis in SE. We break down the abilities needed for artificial intelligence (AI) models to address SE tasks related to code analysis into three categories: 1) syntax understanding, 2) static behavior understanding, and 3) dynamic behavior understanding. Our investigation focused on the ability of LLMs to comprehend code syntax and semantic structures, which include abstract syntax trees (AST), control flow graphs (CFG), and call graphs (CG). We employed four state-of-the-art foundational models, GPT4, GPT3.5, StarCoder and CodeLlama-13b-instruct. We assessed the performance of LLMs on cross-language tasks involving C, Java, Python, and Solidity.

Our findings revealed that while LLMs have a talent for understanding code syntax, they struggle with comprehending code semantics, particularly dynamic semantics. We conclude that LLMs possess capabilities similar to an Abstract Syntax Tree (AST) parser, demonstrating initial competencies in static code analysis. Furthermore, our study highlights that LLMs are susceptible to hallucinations when interpreting code semantic structures and fabricating nonexistent facts. These results indicate the need to explore methods to verify the correctness of LLM output to ensure its dependability in SE. More importantly, our study provides an initial answer to why the codes generated by LLM are usually syntax-correct but vulnerable.

1 INTRODUCTION

ChatGPT [2], released by OpenAI in November 2022, has become one of the most attention-grabbing achievements in the era of AI-Generated Content (AIGC). It is a conversational AI system tuned from a large language model (LLM) from the GPT-3.5 series with Reinforcement Learning from Human Feedback (RLHF) [22, 69, 84]. Later, more large language models are trained and released by the industries and research institutes, like GPT4 [57], StarCoder [42] and LLama2 [11]. The massive learned knowledge of LLMs and the elegant fine-tuning process by RLHF enable these large models to generate high-quality responses to user questions in various domains. Their ability to comprehend context, align instructions, and produce coherent content makes them excellent at multiple tasks such as bilingual translation [62], conversation generation [49] and article summarization [77]. Furthermore, LLM also has attracted widespread attention from the software engineering community as it exhibits excellent capability in software development [55]. It has been widely used in different stages of software development. For example, it can be used in generating code snippets that satisfy the natural language requirements according to the official report [58] by OpenAI. Researchers from the SE community started to explore how to use LLM in SE tasks related to code analysis, for example, Xia and Zhang [86] proposed ChatRepair, which aims to interact with ChatGPT to perform automated program repair in a conversational style. Tian et al. [71] conducted an empirical study to discuss the capability of ChatGPT for code generation, program repair, and code summarization. However, although LLM is widely used and discussed in software engineering, a deep and systematic analysis of LLM's capabilities for code syntax and semantics understanding is vital and worthy of in-depth study.

Program semantics is the essence of a program, and automated learning of program semantics requires the model to comprehend program syntax rules, static behaviors (e.g., data dependencies),

and dynamic behaviors (e.g., execution paths). Hence, program semantics can be regarded as the core for various code-related tasks such as code summarization, code search, and program repair. Pioneering researchers explored the way from different dimensions to accurately learn program semantics such as learning from program structures [17, 47], combining with external knowledge [45, 48], or selecting more powerful neural networks [29, 82]. With the continuous development of techniques, large language models (LLMs) are adopted and they achieved significant progress across various tasks. When the software developer uses these LLMs as programming assistants for their coding tasks, the stunning performance often leads the user to believe that the model has comprehended the program semantics well and the model can produce accurate results. However, *can these LLMs comprehend program semantics?* Some previous works have confirmed that these LLMs are prone to suffer from adversarial attacks [46, 87]. Some simple modifications to the input can mislead the model to produce unexpected outputs. It is unclear whether these LLMs can comprehend program semantics or only copy-paste similar content from the training samples. Moreover, if these LLMs have a certain capability to comprehend program semantics, to what extent can they comprehend the semantics is also unknown.

To address the aforementioned issues, in this paper, we conduct a progressive analysis to explore the capability of LLMs to comprehend program semantics in terms of understanding program syntax, static behaviors, and dynamic behaviors. Our work includes 4 state-of-the-art (SOTA) large language models, GPT4 [57], GPT3.5 [56], StarCoder [42] and CodeLlama-13b-instruct [65]. We design a set of code-related tasks (9 different tasks) on 2,560 code samples. Specifically, for code syntax understanding, we design two tasks, Abstract Syntax Tree (AST) generation and expression matching to find out whether LLM can comprehend program syntax. Besides, we design five tasks including Control Flow Graph (CFG) generation, Call Graph (CG) generation, data dependency analysis, taint analysis, and pointer analysis to explore whether LLM can statically approximate program behavior similar to the traditional static analysis tools [24, 28]. Based on the findings of the understanding ability of LLM on code syntax and static analysis, we further propose two more challenging tasks: equivalent mutant detection and flaky test reasoning to analyze the capability of LLM in dynamically analyzing program behaviors. Through our comprehensive analysis, we found that: (1) LLMs, especially GPT4, are powerful in understanding code syntax. They can understand the syntax role of tokens in the code, and can act as an AST parser; (2) LLMs have certain abilities to analyze the code static behaviors and they can act as a beginner in static analysis; (3) LLMs are limited in approximating dynamic behaviors of the code and thus have poor performance on mutant detection and flaky test reasoning, and we attribute this to a problem with pre-training data; (4) When employing LLM to solve SE tasks, it can suffer from the data-shift problem, that is behaving differently for different projects. We observe that F1 can vary from 0 to 0.8 in the different projects for data dependency and taint analysis. We hope that these findings can better guide software developers in utilizing large language models for software development. Overall, the main contributions of our paper are summarized as follows:

(1) We conduct a comprehensive study from different aspects to explore the capability of LLM for code analysis. We are the first to explore LLM's capability in understanding code syntax, static behaviors, and dynamic behaviors. We study four state-of-the-art models, GPT4, GPT3.5, StarCoder and CodeLlama. We have made our code and data public on our website [6].
(2) We analyze LLM to understand code syntax, code static semantic structures, and code dynamic behaviors through diverse tasks. Our study suggests that LLM is capable of comprehending code syntax rules and it has certain abilities to understand code static behaviors but fails to comprehend dynamic behaviors. GPT4 is the best one to understand code syntax and semantics. Open-source models should be enhanced in code analysis.

2 MOTIVATION

With the strong capabilities of LLMs in coding, such as Copilot [9] and AlphaCode [3], more developers utilize them to recommend the code snippets in daily software development. The recommended content of these tools usually provides senior developers with a great experience [25]. More recently, some tools that rely on LLMs for program repair and testing [68, 85, 86] are proposed. The accurate and satisfactory results produced by these tools seem to be proving to users that these LLMs can learn program semantics well. However, *do they learn program semantics rather than copy-pasting from seen samples?* Since the data used to train LLM are from the Internet across the world, a reasonable assumption is that the queried content is a part of the training samples and LLM only serves as a content distributor to finish copy-paste operation.

To make clear whether LLM can comprehend program semantics or not, a preliminary experiment is conducted for exploration in our study. We use ChatGPT to show our motivation example. Concretely, we used ChatGPT playground [12] with *temperature* of value 0 to perform the conversation in May, 2023. As shown in Figure 1, there is a buggy function "bucketsort" obtained from QuixBugs [13]. The Bucketsort algorithm requires splitting the array (i.e., "arr" in this function) into several buckets (i.e., "counts") and then sorting each bucket individually. Hence, we can find that the correct version to fix this bug function is to replace the variable "arr" in the second loop with the variable "counts". According to an analysis from Sobania et al. [68], ChatGPT can automatically fix this bug as shown in Figure 1. Through this example, it seems that ChatGPT correctly comprehends this function semantics and finishes a correct repair. However, a simple mutation while maintaining the original program semantics can lead ChatGPT to produce incorrect results. As shown in Figure 2, if we change the variable "arr" to another variable name "ccounts" while keeping the other content the same as the former example, ChatGPT cannot fix this buggy function. Hence, the conclusion from both examples is inconsistent. If we just investigate the first example, it seems that ChatGPT can comprehend program semantics and thus successfully fix this buggy function. However, if we further consider the second example, the conclusion is invalid.

As LLM is becoming popular in software engineering (SE) and it plays a significant role in software development, the comprehension of its effectiveness is urgent and significant. Furthermore, understanding the capabilities and limitations of LLM for code analysis is

Does this program have a bug? How to fix it?

```
def bucketsort(arr, k ):
    counts = [0] * k
    for x in arr:
        counts[x] += 1
    sorted_arr = []
    for i, count in enumerate(arr):
        sorted_arr.extend([i] * count )
    return sorted_arr
```

Yes, this program has a bug. The bug is that the variable "counts" is not being used in the second for loop. To fix this bug, the variable "counts" should be used in the second for loop instead of "arr". The corrected code should look like this:

```
def bucketsort(arr, k):
    counts = [0] * k
    for x in arr:
        counts[x] += 1
    sorted_arr = []
    for i, count in enumerate(counts):
        sorted_arr.extend([i] * count)
    return sorted_arr
```

Figure 1: A buggy function of Bucketsort from QuixBugs.

Does this program have a bug? How to fix it?

```
def bucketsort(ccounts, k ):
    counts = [0] * k
    for x in ccounts :
        counts[x] += 1
    sorted_arr = []
    for i, count in enumerate(ccounts):
        sorted_arr.extend([i] * count )
    return sorted_arr
```

Yes, this program has a bug. The variable "ccounts" is used in the first line of the program, but it is not defined anywhere. To fix this bug, the variable "ccounts" should be defined and given a value.

Figure 2: An semantic equivalent version of the buggy function from Figure 1.

important which can ensure that the SE researchers and software developers can use LLM correctly and reasonably for their tasks. To address this challenge, in this paper, we provide a systematic and comprehensive study to investigate the capabilities of LLM for code analysis, i.e., what it can do, and what its limitations are. In this work, we study four large language models, GPT4 [57], GPT3.5 [56], StarCoder [42] and CodeLlama-13b-instruct [65].

3 STUDY DESIGN

3.1 Overview

We begin by examining the abilities that AI models must possess to effectively tackle SE tasks related to code analysis. Generally, through a broad literature review of code modeling works with neural networks, we can categorize existing code models into three different groups. First, some code models [29, 37, 54, 79, 80, 89] are designed to learn code syntax by incorporating abstract syntax tree (AST) information to assist in different SE tasks related to code analysis. Second, researchers also explore the use of control flow and data flow information to enhance code models [15, 32, 50, 81, 90]. Third, there are some works [38, 61, 88] leverage code execution to learn code representations that can address SE tasks related to code semantics. To help the users better understand LLM for code, in this work, we conduct a comprehensive study to explore the capabilities of LLM for code analysis and answer the following three research questions (RQs):

- *RQ1: Can LLM understand code syntax well?*
- *RQ2: Can LLM understand code static behaviors?*
- *RQ3: Can LLM understand code dynamic behaviors?*

Figure 3 demonstrates our analysis framework and illustrates how we estimate the capabilities of LLM for code analysis. The fundamental abilities required to competently perform Software Engineering tasks consist of understanding the syntax of code, the static behaviors of the code, and the dynamic behaviors of the code.

For syntax understanding (RQ1), we first check if LLM can understand code syntax structure (AST) and then design a task that asks LLM to search mathematical expressions that require a good understanding of the syntax roles of tokens. AST is the core structure in code analysis and represents the programming language syntax. We consider using the expression matching task because the expressions are concise for easy analysis, and all the expression tokens have syntax roles. Literally matching without understanding these syntax roles leads to poor performance.

For static understanding (RQ2), the control flow graph (CFG) of code, which estimates the order of statement execution, is considered the first step in the program analysis. Hence, we begin with it and request LLM to construct a CFG from the source code. Call graph (CG) is useful for analyzing large projects by providing the calling context, and we also require LLM to construct CG. Next, we estimate how LLM understands data flow in the code, and conduct data dependency, taint analysis [40], and pointer analysis [67] via LLM. These three tasks reflect the data flow information that is crucial for code analysis. Data flow analysis [39] is an indispensable part of program analysis, and it has significant impacts on programming optimization, bug detection, program understanding, and security analysis.

For dynamic understanding (RQ3), it is interesting to investigate whether LLM can understand how code behaviors change if we make a small modification, which is a more challenging task. Here, we use the equivalent mutant detection task to access the ability of LLM on dynamic understanding. An equivalent mutant refers to a changed program having the same behavior as the original code. It is directly related to the dynamic behavior of the code. We also use flaky test reasoning as a supplement to the equivalent mutant detection. Flaky test reasoning entails explaining why a test sometimes passes and sometimes fails with the same setting. Dynamic behaviors of flaky tests can test how LLM understands the execution of the program with concrete inputs. By investigating LLM's capabilities in these tasks, we can make clear what it can do, as well as what it cannot do. This understanding can help us to figure out what role LLM plays and how to employ LLM in Software Engineering, especially for code analysis tasks.

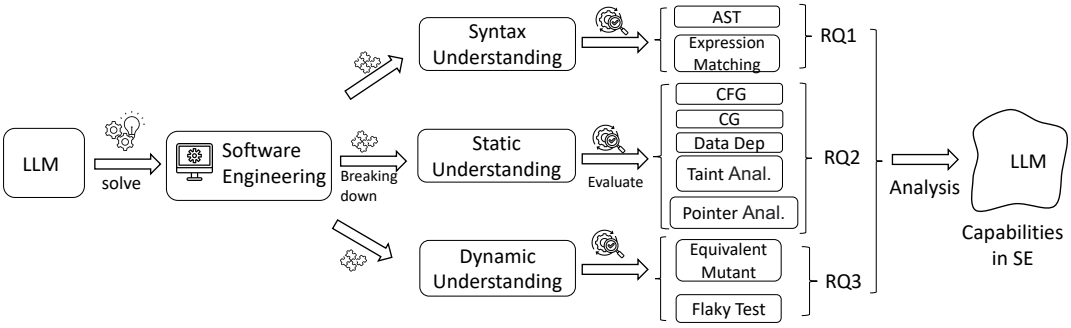


Figure 3: The overview of our study.

3.2 Code Syntax Understanding (RQ1)

Code syntax refers to the set of rules that define valid combinations of symbols in a given programming language. Abstract Syntax Tree (AST) is a data structure that represents code syntax. In this section, our objective is to investigate whether LLM can understand code syntax using AST.

3.2.1 AST Generation. We begin our evaluation by assessing whether LLM can recognize AST structures from the code. AST is widely used in code analysis and also is as the input to solve SE tasks based on learning approaches, such as code clone detection [20] and code representation [89]. We prompt LLM to parse code into an AST, and then compare these ASTs with those generated by programming language AST parsers to determine their meaningfulness. The ability to comprehend ASTs is fundamental for code models, as tokens in code serve distinct syntax roles. Understanding code syntax is crucial for addressing certain SE tasks, such as generating syntactically correct code.

3.2.2 Expression Matching. Then, we explore if LLM can be used for expression matching which is a task that aims to find a similar expression with the target mathematics expression from the input code. This is a task strongly related to code-clone detection for Type-2 and Type-3 that requires understanding the syntax of the code [19, 41]. The matched expression should have almost the same operators as the target expression. Figure 4 presents an example of this task, in which we try to find a similar expression with "base_borrow_rate+utilization_rate*slope1". To accomplish this task, one must understand the operators, operands, and their order in the mathematical expressions defined by the programming language syntax rules. Without understanding the syntax role of tokens, finding similar expressions is not feasible. For instance, "base_borrow_rate-utilization_rate+slope1" may be incorrectly identified as a more similar expression to "base_borrow_rate+utilization_rate*slope1" than to "rate+ur*s1", if the operators "+" and "*" are not recognized. Although the former has variable names similar to the target expression, greater attention should be paid to the syntax-role-operator tokens because they decide the code behaviors. We evaluate whether LLM can leverage its comprehension of code syntax structure to perform an expression matching.

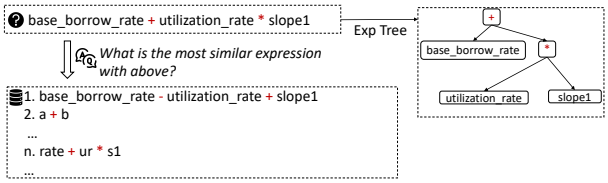


Figure 4: An example of the task of Expression Matching.

3.3 Code Static Behavior Understanding (RQ2)

Static analysis provides fundamental knowledge for solving various code tasks, including bug detection and test generation. Through static analysis, it is possible to identify the relationships among different components of a program, such as the data flow of variables. The objective of this section is to evaluate the performance of LLM in code static behavior analysis, focusing on five types of static behaviors, control flow graph (CFG), call graph (CG), data dependency, taint analysis, and pointer analysis. Task difficulty gradually increases and the later task is related to the previous one.

3.3.1 Control Flow Graph (CFG) Analysis. Control flow graph analysis (CFG) is typically the first step in program analysis and understanding, as it estimates the order of statement execution. To achieve this, we prompt LLM to construct the CFG from a given input code. Figure 5 provides an example ① of this process. Understanding the CFG is critical for code models to identify relationships among statements. CFG is a core code structure in static analysis and is widely employed in software engineering to address various tasks such as vulnerability detection, code optimization, and program analysis [18, 21, 31].

3.3.2 Call Graph (CG) Analysis. The Call Graph is a data structure that depicts the invocation relationship among functions in a program. It is extensively employed in software engineering to understand program behaviors [52], such as interprocedural program analysis. Figure 5 presents an example ② of a call graph with two methods. We prompt LLM to construct the call graph for the given code. Understanding the call graph is significant as it provides insights into the function relationships in the code.

3.3.3 Data Dependency. Figure 6 provides an example ① in which "d" is data-dependent on "a". We prompt LLM to determine whether two given variables are data-dependent in the code. Data dependency analysis is a powerful technique for understanding [32] and

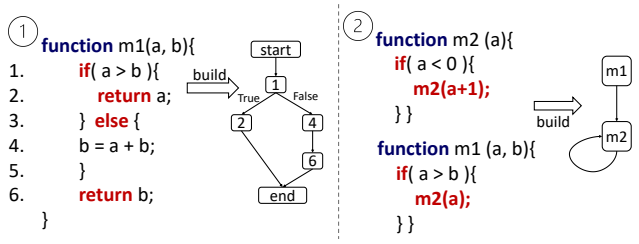


Figure 5: Examples of (1) Code Control Flow Graph (Python) and (2) Code Call Graph (Python).

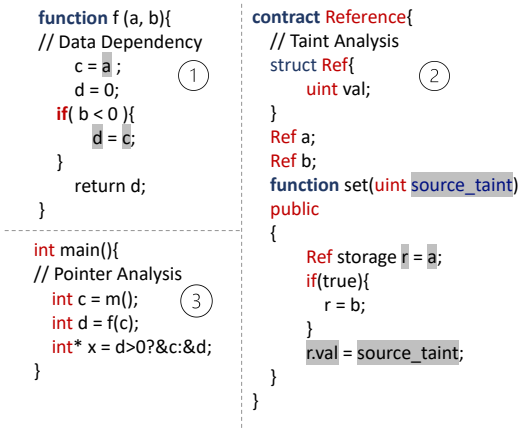


Figure 6: (1) Data Dependency Example (Python), (2) Taint Analysis Example (Solidity) and (3) Pointer Analysis (C).

optimizing code [30], as it can reveal data relationships among different variables in a program. Data dependency illustrates how data are propagated in the program, and it is extremely useful for code models to solve SE tasks such as vulnerability detection [32].

3.3.4 Taint Analysis. In this study, we use taint analysis to detect whether a variable can be tainted by an external source. Figure 6 illustrates an example ② in which the variable "a" can be overwritten by "source_taint" via the storage variable "r". This task necessitates the reasoning ability of LLM based on data dependency analysis. Taint analysis [40] is strongly related to data dependency but also needs information from the call graph and the control flow graph to track how one data point is propagated in the program. It requires a deep understanding and reasoning of the semantics of the code on execution order and relationship.

3.3.5 Pointer Analysis. Pointer analysis is a challenging problem that analyzes the storage positions referred to by pointers. Figure 6 illustrates an example ③ of pointer analysis. The pointer "x" can potentially point to either "c" or "d". Pointer analysis is widely used to detect vulnerabilities such as memory leakage. Pointer analysis [35, 67] requires an understanding of the dependency of data, the control flow, and the call graph. Point analysis also requires the inference to figure out what the current variable refers to. We prompt LLM to infer the referents of pointers. This task requires LLM to comprehend the code syntax and semantics in-depth.

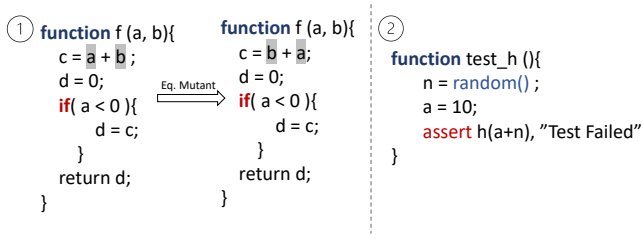


Figure 7: (1) Equivalent Mutant Example (Python) and (2) Flaky Test Reasoning Example (Python).

3.4 Code Dynamic Behavior Understanding (RQ3)

Code syntax and static information are typically the initial inputs required to solve more complex software engineering tasks that involve code dynamic behavior such as code summary. We are interested in if LLM can understand the dynamic behaviors of the code. One effective way is to compare the modified program behavior with the original program and observe the changes in its behavior. Since, in most cases, if we change the code, the code behavior will also change but some mutants are semantically equal. We focus on those mutants that behave consistently with the original program after the changes. It is called equivalent mutant detection in Mutation Testing [60]. We also employ flaky test reasoning. The equivalent mutant task changes the code a little, and then we observe the code's behavior. This flaky test reasoning task is complementary to the former task because it does not change code but can behave differently with the same inputs due to some inside randomness or the external factors such as the latency of the network.

3.4.1 Equivalent Mutant Detection. Detecting equivalent mutants is a critical problem in Mutation Testing [60], which ensures that the mutated code remains functionally equivalent after introducing a random change. Mutation Testing is a technology used to estimate and improve the quality of the test suite by observing how many mutants are detected by the test cases and generating new test cases to identify these injected bugs. Any code change is highly possible to change the dynamic behaviors of the code. This task is consistent with the goal of RQ3. If LLM can really sense the behavior difference due to the minor code change, it can be as evidence that LLM can understand code dynamic behavior well. We prompt LLM to determine whether a mutant is equivalent to the original code. Figure 7 presents an example of an equivalent mutant ①. In this example, we switch the operands "a" and "b" of the addition, but this modification does not affect the result of this function.

3.4.2 Flaky Test Reasoning. Flaky test is one other challenging problem related to code dynamic behavior. Flaky test means the output inconsistency of one test when running multiple times. Thirty flaky reasons are summarized [16]. Flaky tests are usually caused by some undetermined functions, the environment state and the execution schedule. Figure 7 presents one flaky example ② due to randomness. We prompt LLM to tell the reason why one test is flaky.

3.5 Prompt Design

GPT-based models utilize the prompt-based learning paradigm [43]. The design of the prompt can significantly impact the performance of the model. To design better prompts, iteratively, we employ ChatGPT to optimize our initial prompts and then evaluate these optimized prompts by some trial queries. Specifically, we prompt ChatGPT with the message, "*Act as a prompt optimizer and optimize the following prompt for [TASK DESCRIPTION]. The prompt is [PROMPT]*", to help us generate the prompts. [TASK DESCRIPTION] is the task description placeholder and [PROMPT] is the draft prompt placeholder. For each task, we have multiple draft prompts, and then we manually evaluate them using some task data to observe their differences. Finally, according to the experience obtained from the trials, we summarize our prompt templates as *role prompt* and *instruction prompt*. Through our continuous optimizations, our prompt may not be the best, but it is excellent.

Role prompt assigns a specific role to LLM, providing a task context for the model to effectively generate the desired output. Its template is shown below,

```
You are [ROLE] for [LANG]. [TASK DESCRIPTION].
[OUTPUT FORMAT]. The input is [INPUT].
```

where the placeholder [ROLE] denotes the specific role assigned to LLM. We define six roles: AST parser, expression tree matcher, control flow graph analyzer, call graph analyzer, code static analyzer, and pointer analyzer. [LANG] refers to the programming language used for the code analyzed. [TASK DESCRIPTION] outlines the expected task for LLM to perform. [OUTPUT FORMAT] provides the output specification. [INPUT] serves as a placeholder for the code under analysis.

Instruction prompt does not assign a specific role to LLM, instead, they provide a command. These prompts are typically useful for tasks involving multiple roles or those without any applicable roles. The template for the instruction prompt is defined as follows:

```
Please analyze [LANG]. [DOMAIN KNOWLEDGE].
Here are some examples [EXAMPLE CODE]. Please
identify if [TASK DESCRIPTION]. [OUTPUT FORMAT].
The input is [INPUT].
```

where [LANG] specifies the used programming language for the analyzed code. [DOMAIN KNOWLEDGE] explains the domain knowledge relevant to the task. [EXAMPLE CODE] provides sample code related to domain knowledge for task demonstration. [TASK DESCRIPTION] describes the task instruction. [OUTPUT FORMAT] outlines the output specification. [INPUT] serves as a placeholder for the code under analysis. Notice that different LLMs may need different prompt formats. For StarCoder [42] and CodeLlama [65], we also refer to their original papers and adapt the prompt design for both through adding special tokens. For StarCoder, its format is "<|system|>\n<|end|>\n<|user|>\n{query}<|end|>\n<|assistant|>." The placeholder {query} will be replaced by our prompts. For CodeLlama-13b-instruct, we insert the special tokens into our prompt with this format "<s>[INST]{query}[/INST]". For the manual evaluation tasks, we created GPTs tools based on GPT4 [4, 7, 8] to maximize optimization prompt for GPT4.

In our study, we employ the role-based prompt for RQ1 and RQ2 where the prompt does not include examples. Owing to the complexity of these tasks, which entail numerous patterns, prompting the model to generate outputs using illustrated examples might potentially downgrade the performance. For example, by presenting the data-dependence example (Figure 6 ①), the model may hyperfocus on this specific instance, consequently overlooking other situations of data-dependence due to in-context learning [44]. Wang et al. [78] report that few examples in the prompt may not help improve the performance. In contrast, for tasks under RQ3, we resort to instruction prompts given that this type of prompt works a little better during our prompt-designed trials due to their unfitness for resolution by a single role. We present all used prompts on our website [6].

4 EVALUATION SETUP

As LLMs have been trained on the internet data, to avoid the risk of data contamination [83], we utilized new data generated by program analysis tools. We also created a new dataset for Expression Matching using four decentralized finance projects and their whitepapers. The tasks and datasets utilized in our study are summarized in Table 1. We employ 217 programs and extract 2,560 data points for our analysis. The total lines of code is 151,633. In our studies, we employ two closed OpenAI models, GPT4 [57] and GPT3.5 [56], and two open source models, StarCoder [42] and CodeLlama-13b-Instruct [65]. For StarCoder, we use its conversational version, StarChat [73].

4.1 Dataset and Evaluation Metrics

4.1.1 AST Generation. Since programs typically consist of sequential statements, if-else structures, loop structures, try-catch structures, and switch structures, we used small programs that implement these structures. Through the analysis of these basic syntax structures, we can effectively assess the ability of LLM to understand code syntax. Additionally, AST can be redundant, and even a simple program can have a large AST with many nodes, which may lead to output truncation for LLM due to its maximum token handling limitations. We used 75 programs in Python, Java, C and Solodity. We considered the syntax diversity and the whole syntax cheatsheet is presented in our website [6]. To assess LLM's ability to output AST in JSON format, we visualized its outputs for manual evaluation. During the evaluation, we refer to the AST formats from the tree-sitter [14] parser. We classified LLM's output as reasonable or not by analyzing the entire structure with a tolerance for the minor issues (missing trivial leaf nodes); 1) lack leaf nodes but keep the overall structure, we labeled it as 'Yes'. It means that LLM correctly generates the syntax type for the code token but does not give the token itself. 2) If the output provided a wrong structure with incorrect edges or lacked non-leaf nodes, we labeled it as 'No'. In the end, we count how many programs are reasonably handled, and also record and categorize the issues we find.

4.1.2 Expression Matching. This dataset was collected to address a real problem in the blockchain economy. Decentralized finance systems often detail their reward mechanisms in the project whitepaper and make promises that the mechanism is implemented in the project. To create this dataset, we randomly selected four real projects from Ethereum [10], namely ALPHA, BETA, BiFi, and XEN. We implemented 32 reward computing equations strictly based on

Table 1: Tasks and Datasets used in this study.

Task	Level	Language	Evaluation or Comparison	Programs	Dataset Size	LoC
AST	syntax	Python, Java, C, Solidity	Tree-Sitter	75	75	1,059
Expression Matching		Solidity	Top5,10,20	4	32	4,238
CFG	static	Python, Java, C, Solidity	Expert Evaluation	75	75	1,059
CG		Python, Java, C, Solidity	Expert Evaluation	24	24	1,609
Data Dependence		Solidity	Slither	13	992	62,606
Taint Analysis		Solidity	Slither	13	830	62,052
Pointer Analysis		C	Frama-C	40	342	2,726
Flaky Test Reasoning	dynamic	Java	Expert Evaluation	13	65	1,615
Equivalent Mutant Detection		C, Java	Scripts that apply patches	35	200	15728
Total				217	2,560	151,633

their whitepapers and then prompted LLM to find the corresponding implementation in the target Solidity projects. For each query, we fed the target code to LLM as comprehensively as possible and then checked if the corresponding implementation expression was in the top-5, top-10, and top-20 output expressions. For the open source model, CodeLlama and StarCoder, we also computed the cosine similarity between our implemented expressions from the whitepapers and the expressions in the project.

4.1.3 Control Flow Graph (CFG) Generation. Control flow graphs (CFGs) comprise sequence blocks, branch blocks, and loop blocks. To complete this task, we used the dataset from the AST generation task. We visualized the CFGs from LLM for manual evaluation, following the same process as the AST task. We labeled the generated CFGs as reasonable or not by comparing them with their corresponding programs. A CFG was considered reasonable if its overall structure was correct, with tolerating missing 1) start or end nodes, 2) a lack of edges to the end node, or 3) sequence statements being stacked in one node. CFGs that incorrectly represented control structure, 1) wrong branch and loop structures, and 2) fabricated non-existent nodes and edges, were labeled as not reasonable. Finally, we counted the number of reasonable CFGs and recorded and categorized all issues we identified.

4.1.4 Call Graph (CG) Generation. For this task, we selected 24 public source code programs with at least three function calls. These programs, including Python, Java, C, and Solidity, were evaluated using the same method as the AST and CFG generation tasks. A generated call graph (CG) was considered reasonable if all of its call relationships were correct, even with some missing calling or redundant nodes. However, if the output contains non-existent call edges, we think it is not reasonable. The number of correct CGs generated was recorded.

4.1.5 Data Dependency and Taint Analysis. 13 DeFi practical projects were collected from Etherscan for these tasks. We used Slither [1] with 4.1k stars to extract 992 pairs of data-dependency variables and 830 externally tainted variables. To ensure that both datasets were balanced for each project, we downsampled them. Each data sample had one of two labels: 1 indicated that it had a data-dependency fact or could be externally tainted, while 0 indicated that it had no data-dependency or could not be externally tainted. We used F1 as a performance measurement.

4.1.6 Pointer Analysis. We collected 40 C programs that contain different pointer usages that are used as the SVF [70] test suite, which contained 342 pointers. We used Frama-C [24] to extract the possible set of variables for each pointer, which served as the ground truth. For each pointer, we collected a set of variables that it may refer to through prompting LLMs. We used the Jaccard index to measure the similarity between the ground truth set and the predicted set. A Jaccard similar coefficient of 1 indicates that the two sets are identical, while a coefficient of 0 indicates that they are completely dissimilar. We compute the Jaccard index for two scenarios: 1). Jaccard index for each pointer. It can measure how LLM behaves on this task; 2). Jaccard index for each program. It can measure whether LLM has a data-shift problem, that is, behaves differently for different programs.

4.1.7 Equivalent Mutant Detection. We utilized MutantBench [75] for this task. All data information was stored in RDF format, requiring a script to extract the input-label data pairs. The dataset consists of 35 programs in total, and we randomly selected 100 equivalent mutants and 100 nonequivalent mutants. Equivalent mutants are mainly caused by some specific mutant operators [59]. We use the F1 score in this task. Two different prompts were used for this task: one type of prompt did not include any example (zero-shot), and the other type included demonstration examples (few-shot).

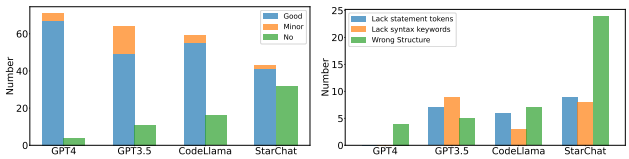
4.1.8 Flaky Test Reasoning. We utilized the publicly available FlakyCat dataset [16]. This dataset, which was manually collected, consists of 13 classes. To create our sample set, we randomly selected 5 samples for each class, resulting in a total of 65 samples. We prompted LLM to assign a label to each input and used accuracy as the performance metric. Similarly to the Equivalent Mutant Detection, we employed two different prompts for this task. We also analyzed the prediction details for each class.

5 EXPERIMENTAL RESULTS

All figures and the generated results for each task by LLMs can be found on our website[6]. We provide statistical analysis in the following sections.

5.1 Code Syntax Understanding (RQ1)

5.1.1 AST Generation. Figure 8a displays the number of reasonable and unreasonable ASTs generated by the four LLMs. Blue bars represent the number of AST, orange bars represent the number of AST with minor issues and green bars represent unreasonable. We can see for GPT4 and GPT3.5, the majority of the generated ASTs



(a) Reasonable VS. Unreasonable (b) Issue Distribution

Figure 8: AST Generation Result

were reasonable and few are minor. However, the open-source models CodeLlama-13-instruct and StarCoder are worse than OpenAI's models and have more unreasonable AST outputs. But CodeLlama is slightly better than StarCoder. We further investigate the issues of the generated ASTs and Figure 8b displays the number of issues that we identified. A single AST may exhibit multiple issues, and even reasonable ASTs might present some minor issues, as explained below. We categorized the found issues into three groups: missing statement tokens (blue bar), missing syntax tokens (orange bar), and wrong structure (green bar) as shown in Figure 8b. The missing-statement-tokens category indicates that some tokens in a statement were missing, such as the token "System" in "System.out.print(a);". The missing-syntax-tokens category indicates that some syntax-related tokens were missing, such as "public" and "private" access modifiers. The wrong-structure category indicates that AST structures were incorrect, such as an incorrect if-else structure. The category Wrong Structure is serious and it means that AST contains faulty syntax structures. Reasonable ASTs with minor issues refer to these ASTs that typically had missing tokens of the statement or missing syntax trivial tokens, such as return type and access modifier. In our evaluation, GPT4 suffers from the least issues. GPT3.5 and CodeLlama have similar performance; GPT3.5 has more missing cases while CodeLlama has more wrong structures. StarCoder is the worst and suffers from lots of wrong structures.

5.1.2 Expression Matching. Table 2 presents the results obtained using the four LLMs. The first column shows the number of ranked equations that were considered in the results. GPT4 and GPT3.5 have a very close hit-rate performance, 27/32 and 28/32 in the top-5, top-10 and top-20. In contrast, StarCoder and CodeLlama have the same hit-rate performance, 5/32. Notice that, the number of StarCoder and CodeLlama in Table 2 are based on the consine similarity. The prompt results of both are quite bad. For StarCoder, none of the outputs is correct. 26/32 outputs are fabricated and 6/32 cases are wrongly matched. For CodeLlama, its prompt outputs only match 1 case and fabricate 28/32 cases.

In our investigation of the answers of GPT4 and GPT3.5, we discovered that both consider two expressions to be similar if they use similar operators, have similar orders, and have a similar number of variables. An example of the target expression and the response from GPT3.5 is shown in Figure 9. The equation above the line represents our target expression, while the response from GPT3.5 is shown below the line. We have highlighted the reasons why GPT3.5 believes that the expressions are similar. We can see that GPT3.5 can recognize multiplication and division operators, as well as related operands. An intriguing observation is that, while GPT3.5 can identify the line number where the matched expression is located or the starting line number of the function containing the matched

```
return (liquidate_amount * price_liquidate /
price_collateral_asset) * liquidation_bonus;
```

```
{
"Answer": "Yes",
"similar_expressions": [
{
"function_name": "calculateCollateralAmount",
"line_number": 107,
"expression": "return (tokenPricePerUnit*_liquidateAmount*
liquidationBonus) / collateralPricePerUnit;"
}
],
"Reason": "Both expressions involve multiplication and
division operations with variables representing prices and
bonuses."
}
```

Figure 9: One Response Example from GPT3.5 about Expression Matching.

Table 2: Expression Matching Results of LLMs

Rank	GPT4	GPT3.5	StarCoder	CodeLlama
Top-5	27	28	4	4
Top-10	27	28	5	5
Top-20	27	28	5	5
Hit Rate	27/32	28/32	5/32	5/32

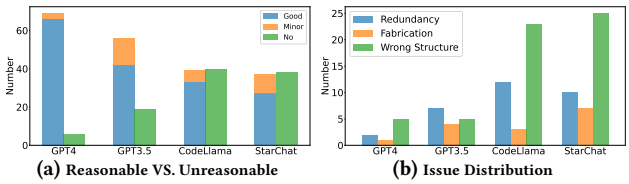
expression, none of the line numbers were accurate for these 32 expression matching samples in either case.

For RQ1: In general, LLM can understand the syntax structure of the code and the syntax roles of the code tokens. This ability allows it to act as an Abstract Syntax Tree (AST) parser. CodeLlama and StarCoder are not as good as OpenAI's models. GPT4 is the best one in the comparison for this research question.

5.2 Code Static Understanding (RQ2)

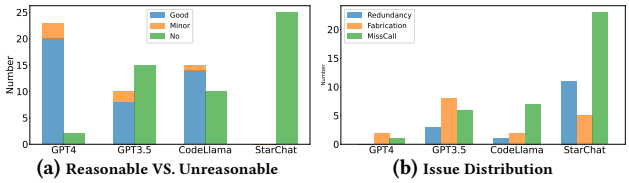
5.2.1 CFG Generation. Figure 10a shows the number of reasonable and unreasonable CFGs generated by ChatGPT according to the predefined criteria. It can be seen that GPT4 achieves the best performance and majority of GPT4's outputs are correct, and few suffer from minor problems or wrong results. GPT3.5 is worse than GPT4 but is better than CodeLlama and StarCoder. CodeLlama and StarCoder have very close performances. Figure 10b shows the issues we identified, which we categorized into three groups: redundancy, fabrication, and wrong structure. One CFG may have multiple issues and reasonable CFGs only can have the redundancy issue. The redundancy category includes meaningless nodes such as null nodes, while the fabrication category contains non-existent nodes or statements. The wrong-structure category refers to CFGs with incorrect structures (incorrectly represented loop statements and if-else statements). Redundancy issues are minor because they do not affect the control flow. Fabrication and wrong-structure issues are serious because they alter the control flow. We observe that GPT4 is still the best one and then the next is GPT3.5. Most of CodeLlama and StarCoder's results suffer from the wrong structure. StarCoder suffers from more hallucinations than others.

Upon examining the identified issues in the AST and CFG generation tasks, an interesting observation is that some serious problems typically arise with loop or if-else statements. LLM appears to have a weaker understanding of the syntax and static behavior of loop and if-else statements.



(a) Reasonable VS. Unreasonable (b) Issue Distribution

Figure 10: CFG Generation Results



(a) Reasonable VS. Unreasonable (b) Issue Distribution

Figure 11: CG Generation Result

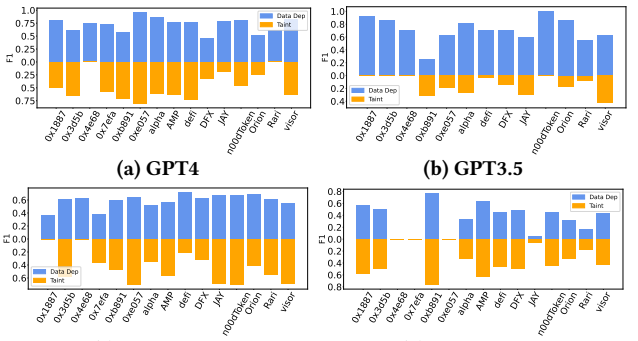
Table 3: Prediction Performance (F1) on Data Dependence and Taint Analysis on the entire datasets.

Task	GPT4	GPT3.5	CodeLLama	StarCoder
Data Dependency	**0.69**	0.68	0.44	0.6
Taint Analysis	0.44	0.15	0.39	**0.47**

5.2.2 Call Graph Generation. Figure 11a shows that GPT3.5, CodeLlama, and StarCoder were unable to generate reasonable CGs for most of the samples while GPT4 still performed well. StarCoder did not generate any reasonable output. Figure 11b illustrates the issues we found and categorized into 3 groups: redundancy, fabrication, and missing call. It can be seen that GPT3.5 suffered from hallucination and GPT4 is obviously better than others. Missing calling is one common issue for the four LLMs. It indicates that GPT4 has a strong ability to understand code semantics.

5.2.3 Data Dependency and Taint Analysis. Table 3 demonstrates the F1 score of the four LLMs on the data dependency task in the second row. GPT4 achieves the best F1 score. GPT3.5 is inferior to GPT4. StarCoder is better than CodeLlama for this task, and worse than OpenAI's models. The comparison of the taint analysis is presented in the last row of Table 3. It can be seen that the performance of LLMs on this task is inferior to the data dependency analysis in terms of F1 (please note that we downsampled the taint dataset to make it balanced). All of them are worse than the random guess classifier that should have a 0.5 F1 score. Taint analysis is based on data dependency and requires the reasoning ability about the data flow. The results show that the studied models lack in-depth reasoning capabilities about the data flow. To assess whether LLM is suffering from the data-shift problem, we conducted an investigation as shown by Figure 12. We computed F1 for each project. Our findings indicate that LLM is significantly affected by the data-shift problem. As illustrated in the upper part (marked in blue) of Figure 12 on the data dependency, F1 scores differ for different projects, with a wide variance ranging from approximately 0. to 1.0. The low part (marked in orange) in Figure 12 shows about F1 scores on taint analysis, which display a variation ranging from 0 to about 0.8.

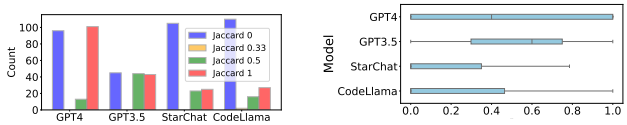
5.2.4 Pointer Analysis. Initially, we determined the number of pointers for which LLMs fully predicted pointer analysis. Out of a



(a) GPT4 (b) GPT3.5

(c) StarCoder (d) CodeLlama

Figure 12: F1 of each project on Data Dependency (blue) and Taint Analysis (orange).



(a) GPT4, Jaccard Index of Each Pointer (b) GPT4, Jaccard Index of Each Pointer

Figure 13: Jaccard Index of Pointer Analysis.

total of 342 pointer samples from the 40 programs, only 105 were correctly predicted by GPT4, 43 were correctly predicted by GPT3.5, 27 and 25 were correctly predicted by CodeLlama and StarCoder respectively. We also noticed that some pointers were missed by LLMs. GPT4 predicts 211/342 pointers, GPT3.5 predicts 133/342 pointers, CodeLlama predicts 156/342 pointers and StarCoder predicts 153/342 pointers. Since one pointer can point to multiple variables, we computed the Jaccard Index between the predicted set and the ground truth set for each pointer, as shown in Figure 13a. We discovered that GPT4 is half good and half bad. It has the largest number for Jaccard Index 1 for each pointer but almost the same number of pointers have Jaccard Index with 0 values. GPT3.5 is inferior to GPT4. CodeLLama and StarCoder are not good in terms of the Jacrad Index for each pointer. We also computed the average Jaccard index for each program to assess whether LLM is affected by the data shift issue. We created a box plot of the mean Jaccard index of pointers from each project, which is illustrated in Figure 13b. We can see that the Jaccard Index variance is quite varied and suggests that LLMs are indeed affected by the data-shift problem.

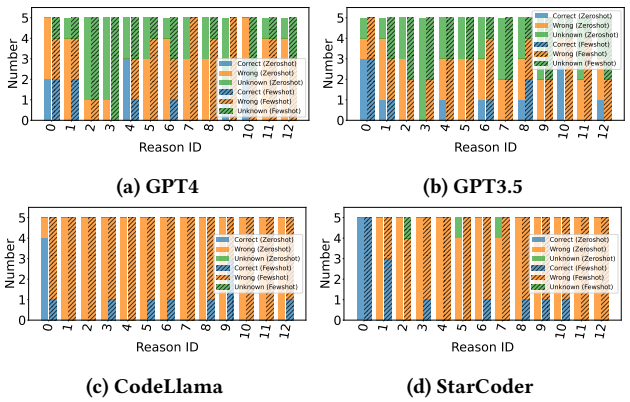
For RQ2: GPT4 and GPT3.5 have the primary ability to perform code static analysis. CodeLlama and StarCoder are not as good as OpenAI's models. However, during the analysis process, we find that all of them experience the issue of hallucination, which can lead to the fabrication of non-existent elements. Furthermore, the performance of LLMs can vary for a given task due to the data shift.

5.3 Code Dynamic Understanding (RQ3)

5.3.1 Equivalent Mutant Detection. Table 4 illustrates the performance F1 score of four LLMs in the detection of equivalent mutants, based on two types of prompts: prompt learning with or without example code (few-shot v.s. zero-shot) because examples are not always helpful to improve model performance [78]. GPT4 and StarCoder perform well. CodeLlama failed to distinguish the equivalent

Table 4: Performance (F1) about Equivalent Mutant Detection.

Type	GPT4	GPT3.5	StarCoder	CodeLlama
few-shot	0.56	0.55	0.57	0.
zero-shot	0.67	0.54	0.62	0.



(a) GPT4 (b) GPT3.5

(c) CodeLlama (d) StarCoder

Figure 14: Predictions of LLMs for Flaky Test Reasoning

mutant and non-equivalent mutant that have one small difference. Even if we tell CodeLlama that the two are equal, it still answers no. Although CodeLlama has the ability to understand code syntax and the limited ability for static analysis as shown in the previous sections, it lacks the reasoning ability for code dynamic behaviors based on the code syntax and static structure understanding. StarCoder demonstrates a better dynamic-understanding ability than CodeLlama due to its pre-training data and tasks. To validate our hypothesis, we investigated the pre-training data and pre-training tasks of StarCoder [42] and CodeLlama [65]. StarCoder uses the code commit data from GitHub while CodeLlama learns publicly available code only without any additional meta-level or temporal information such as git-commit info. Code commit info involves the code behavior change description and its impact.

5.3.2 Flaky Test Reasoning. For this task, we also employed two types of prompts, few-shot and zero-shot prompts. The task comprises 13 classes, with each class containing five samples. To visualize the predicted label number for each class, we used bar figures, which are presented in Figure 14 (bar with slash for few-shot learning and bar without slash for zero-shot learning). Y-axis is the number and X-axis is the reason id. In these figures, the green bar represents the number of predictions that LLM is uncertain about, the orange bar represents the number of incorrect predictions, and the blue bar represents the number of correct predictions. We find that GPT4 and GPT3.5 are more conservative and prefer to answer unknown for most cases, while StarCoder and CodeLlama are quite confident about their outputs. We also find that the demo examples in the prompt can help StarCoder and CodeLlama improve their performance. However, overall, the four models do not perform well for this task.

For RQ3: LLM has limited ability to approximate code dynamic behavior and also suffers from the data shift problem.

6 DISCUSSION

6.1 Limitations

There are several limitations in this study. First, this study does not employ very large datasets for the analysis of each SE task. The reason is that we consider multiple tasks in this study and the dataset preparation is already very expensive. But these tasks are diverse and the analyzed data are created by ourselves via tools, which can help us conduct a comprehensive evaluation of LLMs in various scenarios related to software engineering on code analysis. Second, this study adopts manually designed prompts, however, there are several techniques for automatically designing prompts [66]. It is possible that superior prompts can have better results in this study. We also do not explore how the examples in the prompt affect LLM. In our experiments, we observe that the choice of examples matters and how to find a good example for the prompt is the future independent work. Third, since weights of OpenAI's models are inaccessible, weight analysis commonly used to analyze pre-trained models cannot be used in this study [63] and we can only perform a black analysis. Fourth, as a transformer architecture, LLM imposes a maximum token limitation, restricting the input context in our study. Lastly, we use nine basic and challenging tasks that are related to code analysis in Software Engineering. Some other code analysis tasks are also important but are not included in this study, such as dead code elimination. It may limit our insights to LLMs on code analysis.

6.2 Threats to Validity

The first potential threat lies in the design of the prompts in this work. As LLM's performance will be affected by the designed prompt, to avoid this, we carefully design and optimize the used prompt template. Considering these tasks in our study are domain-related, we add a task and concept explanation to the prompt. We refer to some good prompt engineering repositories [5] to help us design prompts. We plan to explore the effect of different prompts in our future work. The second potential threat lies in the input length of LLM. As the input length for LLMs is limited, to mitigate this problem, we do not use very large programs in the evaluation. Another threat lies in data leakage, as LLM is trained using data from the Internet, it is risky to analyze LLM using the current datasets, especially the well-known ones which may mislead us into wrong conclusions. To mitigate it, we employ the code static analysis tools to create new datasets. These selected program analysis tools are popular and widely validated in different applications.

7 RELATED WORK

Probing Analysis: Probing analysis [64] is a technique employed to examine and interpret the mechanisms of large language models (LLMs). Clark et al.[23] discovered that BERT[26] can encode language syntax by analyzing attention heads. Hewitt et al.[34] proposed a structural probing task to investigate BERT. Betty et al.[74] employed a question-answering (QA) probing task to analyze the reasoning process of BERT. Recently, some works have started to analyze code models, like CodeBERT, CodeT5, and UnixCoder. Wan et al. [76] and Hernández López et al. [33] analyze how code models learn syntax. Troshin and Chirkova [72] and Ma et al. [51] analyze

how code models represent semantics. But all of them study non-foundational models based on tuning classifiers. Our work studies the generation ability of LLM on code analysis.

LLM for SE: Researchers in SE have begun exploring the application of LLM to solve SE problems. Xia et al.[86] proposed a dialogue-style approach for automatic program repair. Dominik et al.[68] conducted a bug-fixing study to evaluate LLM's performance. Tian et al. [71] investigated LLM's capabilities in code generation, program repair, and code summarization. Hou et al. [36] systematically reviews the various applications of language models in software engineering. Fan et al. [27] and Nguyen-Duc et al. [53] discuss how LLM can change software engineering. None of them study the interpretability and reliability of LLM on code analysis, and our work is to estimate how LLM understands code that provides the interpretability of LLM for SE tasks. We are the fist one to study LLM on code analysis, especially for understanding code syntax and semantics, providing some confidence when empoloying LLMs to solve code tasks.

8 CONCLUSION

In this paper, we conduct a comprehensive empirical study to investigate the capabilities of LLM for code analysis. In particular, we study LLM's ability to comprehend code syntax, static behaviors, and dynamic behaviors by *2,560* code samples with *9* different SE tasks. We used the new datasets created by the code tools in our study, including four programming languages: Python, Java, C, and Solidity. Overall, the results of our study indicate that LLM is capable of comprehending code syntax rules and has certain abilities to understand static behaviors of the code, but it does not understand dynamic behaviors well. GPT4 achieves the best performance among all four models we included. We believe that our findings offer insights into LLM's performance on SE tasks related to code analysis and guide follow-up researchers to effectively utilize LLM in the future to solve SE tasks.

9 DATA AVAILABILITY

We publish all data and code of our experiments[6].

REFERENCES

[1] 2018. *Slither.* https://github.com/crytic/slither
[2] 2022-11. *Chatgpt: Optimizing language models for dialogue.* https://chat.openai.com
[3] 2023. *Alphacode.* https://www.deepmind.com/blog/competitive-programming-with-alphacode
[4] 2023. *AST Analyzer.* https://chat.openai.com/g/g-cAZMow3gy-ast-analyzer
[5] 2023. awesome-chatgpt-prompts. https://github.com/f/awesome-chatgpt-prompts.
[6] 2023. *Capabilities of ChatGPT for Code Analysis: An Empirical Study.* https://sites.google.com/view/chatgpt4se
[7] 2023. *CFG Analyzer.* https://chat.openai.com/g/g-rY90G6DgV-cfg-analyst
[8] 2023. *CG Analyzer.* https://chat.openai.com/g/g-P5Qzq5vdB-call-graph-analyzer
[9] 2023. *Copilot.* https://github.com/features/copilot
[10] 2023. *Etherscan.* https://etherscan.io/
[11] 2023. *Llama2.* https://ai.meta.com/llama/
[12] 2023. *Openai Playground.* https://platform.openai.com/playground
[13] 2023. *QuixBugs.* https://github.com/jkoppel/QuixBugs/
[14] 2023. *Tree sitter.* https://tree-sitter.github.io/tree-sitter/
[15] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[16] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2022. Predicting Flaky Tests Categories using Few-Shot Learning. *arXiv preprint arXiv:2208.14799* (2022).
[17] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[18] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479
[19] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* 368–377. https://doi.org/10.1109/ICSM.1998.738528
[20] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* IEEE, 368–377.
[21] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS).* IEEE, 41–50.
[22] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).
[23] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341* (2019).
[24] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.
[25] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.
[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[27] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).
[28] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain* (Montreal, Quebec, Canada) *(WETSEB '19).* IEEE Press, 8–15. https://doi.org/10.1109/WETSEB.2019.00008
[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[30] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1984. The program dependence graph and its use in optimization. In *International Symposium on Programming.* Springer, 125–132.
[31] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041
[32] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[33] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 1–11.
[34] John Hewitt and Christopher D. Manning. 2019. A Structural Probe for Finding Syntax in Word Representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Association for Computational Linguistics, Minneapolis, Minnesota, 4129–4138. https://doi.org/10.18653/v1/N19-1419
[35] Michael Hind and Anthony Pioli. 2000. Which Pointer Analysis Should I Use? *SIGSOFT Softw. Eng. Notes* 25, 5 (aug 2000), 113–123. https://doi.org/10.1145/347636.348916
[36] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).
[37] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence.* PMLR, 54–63.
[38] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer*

and Communications Security (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1631–1645. https://doi.org/10.1145/3548606.3560612

[39] Ken Kennedy. 1979. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division.

[40] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. 2014. Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. IEEE, 269–272.

[41] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.

[42] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[43] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[44] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

[45] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405* (2020).

[46] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning. *arXiv preprint arXiv:2301.09072* (2023).

[47] Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* (2023).

[48] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[49] James H Lubowitz. 2023. ChatGPT, an artificial intelligence chatbot, is impacting medical literature. *Arthroscopy* 39, 5 (2023), 1121–1122.

[50] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. GraphCode2Vec: generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 524–536.

[51] Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiexin Zhang, Wenhan Wang, and Yang Liu. 2022. Are Code Pre-trained Models Powerful to Learn Code Syntax and Semantics? https://api.semanticscholar.org/CorpusID:258556996

[52] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (apr 1998), 158–191. https://doi.org/10.1145/279310.279314

[53] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, et al. 2023. Generative Artificial Intelligence for Software Engineering–A Research Agenda. *arXiv preprint arXiv:2310.18648* (2023).

[54] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2006–2018. https://doi.org/10.1145/3510003.3510096

[55] OpenAI. 2019. ChatGPT Demo. https://www.youtube.com/watch?v=outcGtbnMuQ&ab_channel=OpenAI.

[56] OpenAI. 2023. *GPT-3.5 Turbo fine-tuning and API updates*. https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates

[57] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://api.semanticscholar.org/CorpusID:257532815

[58] OpenAI. 2023. GPT-4 Technical Report. *arXiv* (2023).

[59] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 936–946.

[60] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[61] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[62] Keqin Peng, Liang Ding, Qihuang Zhong, Li Shen, Xuebo Liu, Min Zhang, Yuanxin Ouyang, and Dacheng Tao. 2023. Towards making the most of chatgpt for machine translation. *arXiv preprint arXiv:2303.13780* (2023).

[63] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A Primer in BERTology: What We Know About How BERT Works. *Transactions of the Association for Computational Linguistics* 8 (01 2021), 842–866. https://doi.org/10.1162/tacl_a_00349 arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00349/1923281/tacl_a_00349.pdf

[64] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics* 8 (2021), 842–866.

[65] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[66] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980* (2020).

[67] Yannis Smaragdakis, George Balatsouras, et al. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69.

[68] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[69] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.

[70] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[71] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[72] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates (Hybrid), 371–383. https://aclanthology.org/2022.blackboxnlp-1.31

[73] Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. 2023. Creating a Coding Assistant with StarCoder. *Hugging Face Blog* (2023). https://huggingface.co/blog/starchat.

[74] Betty van Aken, Benjamin Winter, Alexander Löser, and Felix A. Gers. 2019. How Does BERT Answer Questions? A Layer-Wise Analysis of Transformer Representations. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 1823–1832. https://doi.org/10.1145/3357384.3358028

[75] Lars van Hijfte and Ana Oprescu. 2021. MutantBench: an Equivalent Mutant Problem Comparison Framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 7–12. https://doi.org/10.1109/ICSTW52544.2021.00015

[76] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.

[77] Jiaan Wang, Yunlong Liang, Fandong Meng, Zhixu Li, Jianfeng Qu, and Jie Zhou. 2023. Cross-Lingual Summarization via ChatGPT. *arXiv preprint arXiv:2302.14229* (2023).

[78] Weishi Wang, Yue Wang, Steven Hoi, and Shafiq Joty. 2023. Towards Low-Resource Automatic Program Repair with Meta-Learning and Pretrained Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 6954–6968. https://aclanthology.org/2023.emnlp-main.430

[79] Wenhan Wang, Kechi Zhang, Ge Li, Shangqing Liu, Zhi Jin, and Yang Liu. 2022. A Tree-structured Transformer for Program Representation Learning. *arXiv preprint arXiv:2208.08643* (2022).

[80] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[81] Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. 2022. CODE-MVP: learning to represent source code from multiple views with contrastive pre-training. *arXiv preprint arXiv:2205.02029* (2022).

[82] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[83] Tianwen Wei, Liang Zhao, Lichang Zhang, Bo Zhu, Lijie Wang, Haihua Yang, Biye Li, Cheng Cheng, Weiwei Lü, Rui Hu, et al. 2023. Skywork: A more open bilingual foundation model. *arXiv preprint arXiv:2310.19341* (2023).

[84] Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback.

arXiv preprint arXiv:2109.10862 (2021).

[85] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[86] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[87] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.

[88] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

[89] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[90] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).